修 士 論 文

関数型プログラムにおけるプログラム変換の研究

指導教官　外山芳人 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

村島康哲

1999 年 2 月 15 日

目 次

第 1 章

はじめに

1.1 研究の背景・目的

プログラム変換というプログラム作成法は、複雑なプログラムをより効率的に作成するために考案された。プログラムの設計をする時に、プログラマーは効率の良さにとらわれずに理解しやすさ・保守の容易さに重点をおく。そうして作られたプログラムを自動または半自動的方法で変換することで、効率の良いプログラムを得ることができ、ソフトウェアの生産性を高めるものとして期待されている [1, 9]。

変換の概略は図 1.1 のようになる [5]。与えられた初期プログラム P0 に対し、種々の

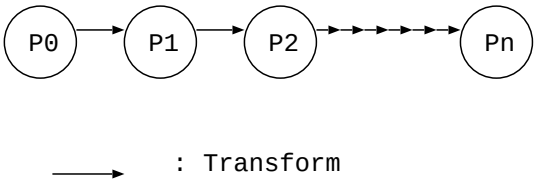

図 1.1: プログラム変換

変換規則を適用して最終的なプログラム Pn を得る。この時、変換途中および最終プログラム（P1, ⋯ Pn）の意味は P0 と等しい。

プログラム変換が注目を浴びるきっかけとなったのは Burstall & Darlington(1977) の研究である。彼らは、与えられたプログラムに変換規則を次々と適用して変換を繰り返すことで、様々なプログラム変換が可能になることを示した [1]。この他、部分計算 (Partial

Evaluation), 上位コンパイラ (Super Compiler) などのプログラム変換法が提案されている [3, 15, 9]。

プログラム変換で、Burstall & Darlington が示した手法のうち定義 (definition)、たたみ込み (folding)、展開 (unfolding) を主たる規則とする戦略は、合成戦略 (composition strategy) と呼ばれる [9]。ここで、定義とは、既に定義された関数を用いて新しい関数を定義することである。また、たたみ込みとは、項の中に出現している一部分がある関数の定義式の右辺と一致しているならば、その部分を左辺の関数に置き換えてまとめる操作である。展開とは、たたみ込みの逆で、項の中に出現している一部分がある関数の定義式の左辺と一致しているならば、 その部分を右辺で展開する操作である。

ここで、本研究の対象となる関数型言語によるプログラムについて概観する。関数型プログラミングは、基本的な関数を定義し、それらの合成によってプログラムを作成する。基本的な関数はリストや木といったデータ構造上で定義されることが多い。関数の合成により、数多くの中間データが作成される。だが、このようなデータ構造を計算途中で生じさせるプログラムは、それらを生じさせないものと比較して一般に計算効率が悪い。

そこで、不必要な中間データを除去することを目的として Wadler(1990) は、切り払い (deforestation) と呼ばれる手法を提案した [16]。切り払いは、7 つの変換規則をプログラムに適用することで実現される。変換規則は、展開、たたみ込みのような従来の規則を含むもので、比較的理解しやすい。また、プログラムに応じた規則を次々に適用するだけで変換が実行されるという単純な処理であるため、変換過程の自動化に適している。しかし、アルゴリズムの効率が不十分であること、変換対象のプログラムの性質が限定されること、高階プログラムを直接扱えない等の問題を含んでいる。

アルゴリズムの効率化に関しては、Wadler 自身が、数値型やブール型なら計算途中に現れても計算効率を悪化させないことに着目し、印付き切り払い (blazed deforestation) という手法を提案している [16]。これは、変換すべきでない項に印をつけ、それらの項を変換対象から外すことにより効率化をはかる手法である。また Sϕrensen(1994)、Seidl(1996) らは、変換によって大幅に増大するパラメータと関数呼び出しの効率化について研究している [14, 13]。また、印付き切り払いでは、数値型やブール型の変数の 2 回以上出現も許している。しかし、木やリストなどの変数は線形に制限されており、プログラムを作成する上で非常に大きな制限になっている。

実装の面では、尾上ら (1997) によりＨＹＬＯシステムと呼ばれるプログラム融合変換

システムが開発、研究されている [8]。ＨＹＬＯシステムでは、遅延評価言語 Gofer で書かれたプログラムを変換の対象としている。

現在までに提案された切り払いの多くは、遅延評価 (lazy evaluation) 言語を対象として行なわれている。切り払いが先行評価 (eager evaluation) に基づく言語を対象に行なわれないのは、変換規則の中に、正当性が保証されない規則がある事が原因である。

佐賀 (1998) は、変換の停止性と、変換前後のプログラムの等価性 (正当性からステップ数の改善に関する性質を除いた性質) が保証された部分計算的切り払いを提案した [10]。部分計算的切り払いは先行評価プログラムを対象とした切り払いで、切り払いが先行評価言語においても有効であることを示している。しかし、ステップ数の改善に関して、理論的な考察はなされていない。

そこで、本研究では、Wadler が提案した印付き切り払いをもとに、先行評価言語における切り払いの変換の正当性に影響を与える問題点を検討する。そして、リストや木などのデータ構造に関しても線形に制限しないで良いような切り払いを提案する。さらに、先行評価言語を対象とした場合、中間データを積極的に作ることによって効率を良くすることも可能であることも示す。

1.2 構成

第 2 章では必要となる用語の解説を行なう。続いて第 3 章では、印付き切り払いの先行評価における問題点とその解決法を示す。第 4 章では第 3 章で示した問題点を解決した先行評価言語向けの切り払いを提案する。最後に第 5 章で本研究を概観し、今後の研究方向について言及する。

第 2 章

準備

本章では、[16, 6] に準じて本稿に必要となるプログラム、プログラム変換に関する諸定義を与える。

2.1 言語の定義

定義 2.1.1 (項) 変数記号、関数記号、構成子記号をそれぞれ v, c, f と表すとき、項 t を以下のように定義する。

$$
\begin{array}{rcll}
t & ::= & v & (\text{変数}) \\
 & | & c(t_1, \ldots, t_k) & (\text{構成子式}) \\
 & | & f(t_1, \ldots, t_k) & (\text{関数式}) \\
 & | & pr(t_1, \ldots, t_k) & (\text{原始関数}) \\
 & | & \mathbf{case}\ t_0\ \mathbf{of}\ p_1\ :\ t_1 \,|\, \ldots \,|\, p_n\ :\ t_n & (\mathbf{case}\ \text{式}) \\
p & \stackrel{def}{=} & c(v_1, v_2, \ldots, v_k) & (\text{パターン})
\end{array}
$$

但し、**case** 式の $k \geq 1$ とし、$p_x \Rightarrow\ t_x$を分岐 (branch) と呼ぶ。

定義 2.1.2 (再帰的データ型) $'a$ を型変数 (type variable) とする。このとき以下のように定義されるデータ型を再帰的データ型と呼ぶ。

$$
\begin{array}{rcll}
'a\ data & \stackrel{def}{=} & c_1 & (\text{基底データ}) \\
 & | & c_2('a_1, \ldots, '\ a_l, '\ a\ data_1, \ldots, '\ a\ data_m) & (\text{帰納データ})
\end{array}
$$

但し $m \geq 1$ とし、帰納データの引数$'a_1, \ldots, '\ a_l, '\ a\ data_1, \ldots, '\ a\ data_m$の位置は任意。

例 2.1.1 (再帰的データ型（１）) リストは、再帰的データ型として次のように表現できる。

$$\begin{aligned} 'a\ list\ &=\ \ Nil \\ &\quad \mid Cons('a,\ 'a\ list) \end{aligned}$$

例 2.1.2 (再帰的データ型（２）) 木は、再帰的データ型として次のように表現できる。

$$\begin{aligned} 'a\ tree\ &=\ \ Leaf(z) \\ &\quad \mid Branch('a\ tree,\ 'a\ tree) \end{aligned}$$

定義 2.1.3 (関数定義、プログラム) 関数 fは、以下のように項 tで定義される。特に **case** 式で定義される場合、その定義を **case** 式による関数定義と呼ぶ。また、関数定義の集合をプログラムと呼ぶ。

$$f(v_1, v_2, \ldots, v_k) = t$$

定義 2.1.4 (再帰関数) 再帰関数は **case** 式によってのみ定義される。

定義 2.1.5 (分岐変数) **case** 式による関数定義において、そのパターンが再帰的データ型である次のような関数定義を考える。但し、c_1 は基底データ、$c_2(\mathbf{v_d}, \mathbf{v_D})$ は帰納データ、$\mathbf{v_d} = (v_{d_1}, v_{d_2}, \ldots, v_{d_l})$ は帰納データにおけるデータの要素を表す変数列、$\mathbf{v_D} = (v_{D_1}, v_{D_2}, \ldots, v_{D_m})$ は帰納データにおける再帰的データを表す変数列とする。

$$\begin{aligned} f(v_1, v_2 \ldots, v_k, v_{k+1}, v_{k+2}) = \mathbf{case}\ v_{k+2}\ \mathbf{of} \\ c_1 \Rightarrow t_1 \\ \mid c_2(\mathbf{v_d}, \mathbf{v_D}) \Rightarrow t_2 \end{aligned}$$

このとき v_{k+2}のように **case** 式の判定対象となる変数を分岐変数と呼ぶ。

本研究では、先行評価言語を対象としている。ここで、一階の操作的意味論に基づいて式の意味を定義する。また、先行評価と比較するために遅延評価の意味も定義する。

定義 2.1.6 (値の定義) 値 *(value)* を次のように定義する。
Vが定数か変数の時、Vは値である。
部分項 $t_1, \ldots, t_k$が値の時、構成子式 $c(t_1, \ldots, t_k)$ は値である。

定義 2.1.7 (先行評価の簡約規則) 式の評価は次の規則によって評価する。先行評価での *1* ステップの簡約を$\overset{eager}{\rightarrow}$で表す。

1. 公理

- $f(t_1, \ldots, t_k) \overset{eager}{\rightarrow} [t_1/x_1, \ldots, t_k/x_k]t$
  ($t_1, ...t_k$ が値で、$f(x_1, ...x_k) = t$ と定義されていると仮定する。)
- $\mathbf{case}\ V\ \mathbf{of}\ p_1 \Rightarrow t_1 \mid p_2 \Rightarrow t_2 \mid \ldots \mid p_k \Rightarrow t_k \overset{eager}{\rightarrow} [V_1'/x_1, \ldots, V_k'/x_k]e_i$
  (Vが値でかつ、$V = c_i(V_i', \ldots, V_k'), pat_i = c_i(x_1, \ldots, x_k)$ と仮定する)
- $\mathbf{let}\ u_1 = V\ \mathbf{in}\ M \overset{eager}{\rightarrow} [V/u_1]M$ （Vが値である）
- $0 + 0 \overset{eager}{\rightarrow} 0,\ 0 + 1 \overset{eager}{\rightarrow} 1,\ \ldots,\ 3 + 5 \overset{eager}{\rightarrow} 8, \ldots$
- $0 * 0 \overset{eager}{\rightarrow} 0,\ 0 * 1 \overset{eager}{\rightarrow} 0,\ \ldots,\ 3 * 5 \overset{eager}{\rightarrow} 15,\ \ldots$

  以下の *5* つの規則では *x,y* が共に値であると仮定する。
- $$x \wedge y \overset{eager}{\rightarrow} \begin{cases} true & x \text{ と } y\text{が共に真である} \\ false & \text{その他} \end{cases}$$
- $$x \vee y \overset{eager}{\rightarrow} \begin{cases} false & x \text{ と } y\text{が共に偽である} \\ true & \text{その他} \end{cases}$$
- $$x \leq y \overset{eager}{\rightarrow} \begin{cases} true & x \text{ が } y\text{より大きいか等しい} \\ false & \text{その他} \end{cases}$$
- $$x > y \overset{eager}{\rightarrow} \begin{cases} true & y\text{が } x \text{ よりも小さい} \\ false & \text{その他} \end{cases}$$
- $$\neg x \overset{eager}{\rightarrow} \begin{cases} true & x \text{ が偽である} \\ false & x \text{ が真である} \end{cases}$$

2. 部分項

- 構成子式

$$\frac{M_1 \overset{eager}{\rightarrow} M_1'}{c(M_1,\ldots,M_k) \overset{eager}{\rightarrow} c(M_1',\ldots,M_k)}$$

$$\frac{M_i \overset{eager}{\rightarrow} M_i'}{\begin{array}{l} c(V_1,\ldots,V_{i-1},M_i,M_{i+1},\ldots,M_k) \\ \qquad \overset{eager}{\rightarrow} c(V_1,\ldots,V_{i-1},M_i',M_{i+1},\ldots,M_k) \end{array}}$$

($V_1,\ldots,V_{i-1}$が値であると仮定する)

- 関数式

$$\frac{M_1 \overset{eager}{\rightarrow} M_1'}{f(M_1,\ldots,M_k) \overset{eager}{\rightarrow} f(M_1',\ldots,M_k)}$$

$$\frac{M_i \overset{eager}{\rightarrow} M_i'}{\begin{array}{l} f(V_1,\ldots,V_{i-1},M_i,M_{i+1},\ldots,M_k) \\ \qquad \overset{eager}{\rightarrow} f(V_1,\ldots,V_{i-1},M_i',M_{i+1},\ldots,M_k) \end{array}}$$

($V_1,\ldots,V_{i-1}$が値であると仮定する)

- *case* 式

$$\frac{M \overset{eager}{\rightarrow} M'}{\begin{array}{l} \mathbf{case} M \mathbf{of}\, p_1 \Rightarrow t_1 \mid p_2 \Rightarrow t_2 \mid \ldots \mid p_k \Rightarrow t_k \\ \qquad \overset{eager}{\rightarrow} \mathbf{case} M' \mathbf{of}\, p_1 \Rightarrow t_1 \mid p_2 \Rightarrow t_2 \mid \ldots \mid p_k \Rightarrow t_k \end{array}}$$

- *let* 式

$$\frac{M \overset{eager}{\rightarrow} M'}{\mathbf{let} u_1 = M \mathbf{in}\, N \overset{eager}{\rightarrow} \mathbf{let} u_1 = M' \mathbf{in}\, N}$$

定義 2.1.8 (遅延評価の簡約規則) 式の評価は遅延評価では次の規則によって評価する。遅延評価での *1* ステップの簡約を$\overset{lazy}{\rightarrow}$で表す。

1. 公理

- $f(t_1,\ldots,t_k) \overset{lazy}{\rightarrow} [t_1/x_1,\ldots,t_k/x_k]t$
  ($f(x_1,\ldots x_k) = t$ と定義されていると仮定する。)
- $\mathbf{case}\ V\ \mathbf{of}\ p_1 \Rightarrow t_1 \mid p_2 \Rightarrow t_2 \mid \ldots \mid p_k \Rightarrow t_k \overset{lazy}{\rightarrow} [V_1'/x_1,\ldots,V_k'/x_k]e_i$
  ($V = c_i(V_i',\ldots,V_k'), pat_i = c_i(x_1,\ldots,x_k)$ と仮定する)

- $\textbf{let}\ u_1\ =\ V\ \textbf{in}\ M\ \overset{lazy}{\rightarrow}\ [V/u_1]M$
- $0+0\overset{lazy}{\rightarrow}0,\ 0+1\overset{lazy}{\rightarrow}1,\ \ldots,\ 3+5\overset{lazy}{\rightarrow}8,\ldots$
- $0*0\overset{lazy}{\rightarrow}0,\ 0*1\overset{lazy}{\rightarrow}0,\ \ldots,\ 3*5\overset{lazy}{\rightarrow}15,\ \ldots$

以下の 5 つの規則では x,y が共に値であると仮定する。

- $$x\wedge y\overset{lazy}{\rightarrow}\begin{cases} true & x\ \text{と}\ y\text{が共に真である} \\ false & \text{その他} \end{cases}$$
- $$x\vee y\overset{lazy}{\rightarrow}\begin{cases} false & x\ \text{と}\ y\text{が共に偽である} \\ true & \text{その他} \end{cases}$$
- $$x\leq y\overset{lazy}{\rightarrow}\begin{cases} true & x\ \text{が}\ y\text{より大きいか等しい} \\ false & \text{その他} \end{cases}$$
- $$x>y\overset{lazy}{\rightarrow}\begin{cases} true & y\text{が}\ x\ \text{よりも小さい} \\ false & \text{その他} \end{cases}$$
- $$\neg x\overset{lazy}{\rightarrow}\begin{cases} true & x\ \text{が偽である} \\ false & x\ \text{が真である} \end{cases}$$

2. 部分項

- 原始関数

$$\frac{M\overset{lazy}{\rightarrow}M'}{pr(M,N)\overset{lazy}{\rightarrow}pr(M',N)}$$

$$\frac{N\overset{lazy}{\rightarrow}N'}{pr(V,N)\overset{lazy}{\rightarrow}pr(V,N')}$$

（Vは値であると仮定する）

- 構成子式

$$\frac{M_1 \overset{lazy}{\rightarrow} M_1'}{c(M_1,\ldots,M_k) \overset{lazy}{\rightarrow} c(M_1',\ldots,M_k)}$$

$$\frac{M_i \overset{lazy}{\rightarrow} M_i'}{\begin{array}{l} c(V_1,\ldots,V_{i-1},M_i,M_{i+1},\ldots,M_k) \\ \qquad \overset{lazy}{\rightarrow} c(V_1,\ldots,V_{i-1},M_i',M_{i+1},\ldots,M_k) \end{array}}$$

（$V_1,\ldots,V_{i-1}$が値であると仮定する）

- *case* 式

$$\frac{M \overset{lazy}{\rightarrow} M'}{\begin{array}{l} \mathbf{case} M \mathbf{of}\ p_1 \Rightarrow\ t_1 \mid p_2 \Rightarrow\ t_2 \mid \ldots \mid p_k \Rightarrow\ t_k \\ \qquad \overset{lazy}{\rightarrow} \mathbf{case} M' \mathbf{of}\ p_1 \Rightarrow\ t_1 \mid p_2 \Rightarrow\ t_2 \mid \ldots \mid p_k \Rightarrow\ t_k \end{array}}$$

定義 2.1.9 （簡約の記法）$\overset{n}{\rightarrow}, \overset{\leq n}{\Rightarrow}$を次のように定義する。

$t \overset{n}{\rightarrow} t'$ ：tを評価して n ステップで t'にたどり着く。

$t \overset{\leq n}{\Rightarrow} t'$ ：tを評価して n ステップ以下で t'にたどり着く。

例 2.1.3 （項の簡約）先行評価と遅延評価の項の簡約の違いについて、いくつかの例をもって示す。

$f(x) \ = \ x+3$

$ff(x,y) = x$

$g(x,y) \ = \ \mathbf{case}\ x\ \mathbf{of}$

$\qquad\qquad true : y$

$\qquad\qquad false : f(2)$

$gg(x) \ = \ \mathbf{case}\ x > 0\ \mathbf{of}$

$\qquad\qquad true : gg(x+1)$

$\qquad\qquad false : false$

$h(x) = h(x+1)$

と定義する。この時、

- **case** $true$ **of** $\overset{eager}{\rightarrow} 3-2 \overset{eager}{\rightarrow} 1$（*2*ステップ）

  $true\ :\ 3-2$

  $false\ :\ 4-2$

  case $true$ **of** $\overset{lazy}{\rightarrow} 3-2 \overset{lazy}{\rightarrow} 1$（*2*ステップ）

  $true\ :\ 3-2$

  $false\ :\ 4-2$

- $f(5) \overset{eager}{\rightarrow} 5+3 \overset{eager}{\rightarrow} 8$ （*2*ステップ）

  $f(5) \overset{lazy}{\rightarrow} 5+3 \overset{lazy}{\rightarrow} 8$ （*2*ステップ）

- $g(false, f(7)) \overset{eager}{\rightarrow} g(false, 7+3) \overset{eager}{\rightarrow} g(false, 10) \overset{eager}{\rightarrow}$ **case** $false$ **of**

  $true\ :\ 10$

  $false : f(2)$

  $\overset{eager}{\rightarrow} f(2) \overset{eager}{\rightarrow} 2+3 \overset{eager}{\rightarrow} 5$ （*6*ステップ）

  $g(false, f(7)) \overset{lazy}{\rightarrow}$ **case** $false$ **of** $\overset{lazy}{\rightarrow} f(2) \overset{lazy}{\rightarrow} 2+3 \overset{lazy}{\rightarrow} 5$

  $true\ :\ f(7)$

  $false : f(2)$

  （4 ステップ）

- $ff(true, gg(3)) \overset{eager}{\rightarrow} ff(true,$ **case** $3 > 0$ **of**

  $true\ :\ gg(3+1)$

  $false\ :\ false)$

  $\overset{eager}{\rightarrow} ff(true, gg(3+1)) \overset{eager}{\rightarrow} ff(true, gg(4)) \overset{eager}{\rightarrow} \ldots$（停止しない）

  $ff(true, gg(3)) \overset{lazy}{\rightarrow} true$（停止する）

- $h(3) \overset{eager}{\rightarrow} h(3+1) \overset{eager}{\rightarrow} h(4) \overset{eager}{\rightarrow} \ldots$ （停止しない）

  $h(3) \overset{eager}{\rightarrow} h(3+1) \overset{eager}{\rightarrow} h(3+1+1) \overset{eager}{\rightarrow} \ldots$ （停止しない）

2.2 プログラムの変換

プログラム変換においては、変換前のプログラムと、変換後のプログラムの意味が変わらないことが重要である。さらに、計算の効率が良くなることも重要である。ここでは、変換の正当性についての諸定義を行なう。

定義 2.2.1 (プログラムの変換) 項 t を変換するということを $T[\![t]\!]$ と表す。以後、断りのない限り $T[\![t]\!]$ のことを $[\![t]\!]$ で省略する。

定義 2.2.2 *2* 項関係 $t \simeq t'$, $t \trianglerighteq t'$を次のように定義する。
$t \simeq t' \Leftrightarrow t$ の評価が停止しない$\rightarrow t'$ の評価も停止しない
$t \sqsubseteq t' \Leftrightarrow (t \xrightarrow{n} p \Rightarrow t' \overset{\leq n}{\Rightarrow} p)$ かつ $t \simeq t'$

定義 2.2.3 (変換の正当性) 項 t の変換 $[\![t]\!]$ を考える。このが正当性を満たすとは次のことを満たす時にいう。
任意の基底代入σに対し、
$t\sigma \sqsubseteq [\![t]\!]\sigma$の関係を満たす。

以後、$t\sigma \sqsubseteq [\![t]\!]\sigma$の関係を $t \trianglerighteq [\![t]\!]$ と表すことにする。

2.3 印付き切り払い

計算途中に現れる全ての中間データが効率を悪くするとは限らない。数値型やブール型なら計算途中に現れても計算効率を悪化させない。よって、これらの変換すべきでない項に印を付けそれらの項を変換対象から外すことによって効率化をはかる。

定義 2.3.1 (関数定義) 切り払いの対象となる関数は以下のように項 tt で定義される。
$f(x_1, \ldots, x_n) = tt$

$$
\begin{array}{rcl}
tt & ::= & vv \\
& | & (c(tt_1, \ldots, tt_k))^{\oplus} \\
& | & (f(vv_1, \ldots, vv_k))^{\oplus} \\
& | & (\mathbf{case}\ vv_0\ \mathbf{of}\ p_1\ :\ tt_1 \,|\, \ldots \,|p_n\ :\ tt_n)^{\oplus} \\
vv & ::= & v \\
& | & c^{\ominus} \\
& | & (f(vv_1, \ldots, vv_k))^{\ominus} \\
& | & (\mathbf{case}\ vv_0\ \mathbf{of}\ p_1\ :\ vv_1 \,|\, \ldots \,|p_n\ :\ vv_n)^{\ominus}
\end{array}
$$

原始関数は $(f(vv_1, \ldots, vv_k))^{\ominus}$ に含まれる。

上の定義において数値型、ブール型の項が$\ominus$に印付けられる。それ以外の項は$\oplus$に印付けられる。よって、すべての項は$\oplus$か$\ominus$ に印付けられる。また、線形の制限については、次のようにする。

定義 2.3.2 (線形の制限) 関数定義の右辺において、$\oplus$に印付けられた項を表す変数は線形に制限する。そして、$\ominus$に印付けられた項を表す変数は線形でなくても良い。

例 2.3.1 (印付き切り払いにおける関数の例) 以下のような関数がある。

$$
\begin{array}{rcl}
sum & : & \text{リストの要素の総和を求める。} \\
sum & : & int\ list\ \rightarrow\ int \\
sum\ xs & = & sum'\ 0\ xs \\
 & & \\
sum' & : & \text{リストの要素の総和に } a \text{ を加えた数を求める。} \\
sum' & : & int\ \rightarrow\ int\ list\ \rightarrow\ \ int \\
sum'\ a\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Nil\ : a \\
 & & \quad Cons(x, xs)\ : sum'\ (a + x)\ xs \\
 & & \\
squares & : & \text{リストの各要素を 2 乗する。} \\
squares & : & int\ list\ \rightarrow\ int\ list \\
squares\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Nil\ : Nil \\
 & & \quad Cons(x, xs)\ : Cons\ (square\ x)\ (squares\ xs) \\
 & & \\
upto & : & m \text{ から } n \text{ までのリストを作成する。} \\
upto & : & int\ \rightarrow\ int\ \rightarrow\ \ int\ list \\
upto\ m\ n & = & \textbf{case}\ (m > n)\ \textbf{of} \\
 & & \quad True\ :\ Nil \\
 & & \quad False\ :\ Cons\ m\ (upto\ (m + 1)\ n)
\end{array}
$$

定義 2.3.3 (変換規則) 印付き切り払いは次の変換規則を適用することによってプログラムを変換する。

(1) $[\![v]\!] \;=\; v$

(2) $[\![c(t_1,\ldots,t_k)]\!] \;=\; c([\![t_1]\!],\ldots,[\![t_k]\!])$

(3) $[\![f(t_1,\ldots,t_k)]\!] \;=\; [\![t[t_1/x_1,\ldots,t_k/x_k]]\!]$
where f *is defined by* $f(x_1,\ldots,x_k)=t$

(4) $[\![\mathbf{case}\ v\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n]\!] = \mathbf{case}\ v\ \mathbf{of}\ p'_1:[\![t'_1]\!]|\ldots|p'_n:[\![t'_n]\!]$

(5) $[\![\mathbf{case}\ c(t_1,\ldots,t_k)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n]\!] = [\![t'_i[t_1/v_1,\ldots,t_k/v_k]]\!]$
where $p'_i \;=\; c(v_1,\ldots,v_k)$

(6) $[\![\mathbf{case}\ f(t_1,\ldots,t_k)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n]\!]$
$= [\![\mathbf{case}\ t[t_1/v_1,\ldots,t_k/v_k]\ of\ p'_1:t'_1|\ldots|p'_n:t'_n]\!]$
where f *is defined by* $f(v_1,\ldots,v_k)=t$

(7) $[\![\mathbf{case}\ (\mathbf{case}\ v\ \mathbf{of}\ p_1:t_1|\ldots|p_m:t_m)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n]\!]$
$= [\![\ \mathbf{case}\ v\ \mathbf{of}$
$p_1\ :\ (\mathbf{case}\ t_1\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n)$
$p_m\ :\ (\mathbf{case}\ t_m\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n)]\!]$

(8) $[\![f(t_1^{\ominus},\ldots,t_k^{\ominus})^{\ominus}]\!] \;=\; f([\![t_1]\!],\ldots,[\![t_k]\!])$

(9) $[\![\mathbf{case}\ f(t_1^{\ominus},\ldots,t_k^{\ominus})^{\ominus}\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n]\!]$
$= \mathbf{case}\ f([\![t_1]\!],\ldots,[\![t_k]\!])\ \mathbf{of}\ p'_1:[\![t'_1]\!]|\ldots|p'_n:[\![t'_n]\!]$

(10) $[\![\mathbf{let}\ v^{\ominus} \;=\; t_0^{\ominus}\ \mathbf{in}\ t_1]\!]$
$= \mathbf{let}\ v^{\ominus} \;=\; [\![t_0^{\ominus}]\!]\ \mathbf{in}\ [\![t_1]\!]$

(11) $[\![\mathbf{case}\ (\mathbf{let}\ v^{\ominus} \;=\; t_0^{\ominus}\ \mathbf{in}\ t_1)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n]\!]$
$= \mathbf{let}\ v^{\ominus} \;=\; [\![t_0^{\ominus}]\!]\ \mathbf{in}\ [\![\mathbf{case}\ t_1\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n]\!]$

例 2.3.2 $squares(upto\ 1\ n)$ の変換の過程を以下に示す。

$[\![squares(upto\ 1\ n)]\!]$
$= [\![\mathbf{let}\ u_1=1\ \mathbf{in}\ squares(upto\ u_1\ n)]\!]$
$= \mathbf{let}\ u_1=[\![1]\!]\ \mathbf{in}\ [\![squares(upto\ u_1\ n)]\!]$
$= \mathbf{let}\ u_1=1\ \mathbf{in}\ [\![squares(upto\ u_1\ n)]\!]$ *—(1)*

$$
\begin{aligned}
&[\![squares(upto\ u_1\ n)]\!] \\
&= [\![\textbf{case}\ \ (upto\ u_1\ n)\ \textbf{of} \\
&\qquad\qquad nil\ :\ nil \\
&\qquad\qquad cons(x, xs)\ :\ cons\ (square\ x)\ (squares\ xs)]\!] \\
&= [\![\textbf{case}\ \ (\textbf{case}\ (u_1 > n)\ \textbf{of} \\
&\qquad\qquad\qquad true\ :\ nil \\
&\qquad\qquad\qquad false\ :\ cons\ u_1\ (upto\ u_1 + 1\ n))\ \textbf{of} \\
&\qquad\qquad nil\ :\ nil \\
&\qquad\qquad cons(x, xs)\ :\ cons\ (square\ x)\ (squares\ xs)]\!] \\
&= [\![\textbf{case}\ u_1 > n\ \textbf{of} \\
&\qquad\quad true\ :\ \textbf{case}\ nil\ \textbf{of} \\
&\qquad\qquad\qquad\qquad nil\ :\ nil \\
&\qquad\qquad\qquad\qquad cons(x, xs)\ :\ cons\ (square\ x)\ (squares\ xs)]\!] \\
&\qquad\quad false\ :\ \textbf{case}\ cons\ u_1\ (upto\ u_1 + 1\ n)\ \textbf{of} \\
&\qquad\qquad\qquad\qquad nil\ :\ nil \\
&\qquad\qquad\qquad\qquad cons(x, xs)\ :\ cons\ (square\ x)\ (squares\ xs)]\!] \\
&= \textbf{case}\ [\![u_1]\!] > [\![n]\!]\ \textbf{of} \\
&\qquad true\ :\ [\![\textbf{case}\ nil\ \textbf{of} \\
&\qquad\qquad\qquad\qquad nil\ :\ nil \\
&\qquad\qquad\qquad\qquad cons(x, xs)\ :\ cons\ (square\ x)\ (squares\ xs)]\!] \\
&\qquad false\ :\ [\![\textbf{case}\ cons\ u_1\ (upto\ u_1 + 1\ n)\ \textbf{of} \\
&\qquad\qquad\qquad\qquad nil\ :\ nil \\
&\qquad\qquad\qquad\qquad cons(x, xs)\ :\ cons\ (square\ x)\ (squares\ xs)]\!] \\
&= \textbf{case}\ u_1 > n\ \textbf{of} \\
&\qquad true\ :\ [\![nil]\!] \\
&\qquad false\ :\ [\![cons\ (square\ u_1)\ (squares\ (upto\ u_1 + 1\ n))]\!] \\
&= \textbf{case}\ u_1 > n\ \textbf{of} \\
&\qquad true\ :\ nil \\
&\qquad false\ :\ cons\ [\![(square\ u_1)]\!]\ [\![(squares\ (upto\ u_1 + 1\ n))]\!] \\
&= \textbf{case}\ u_1 > n\ \textbf{of}
\end{aligned}
$$

$$
\begin{aligned}
&\quad true \ : \ nil \\
&\quad false \ : \ cons \ \ (square\ [\![u_1]\!])\ [\![\mathbf{let} u_2 = u_1 + 1 \mathbf{in} (squares\ (upto\ u_2\ n))]\!] \\
&= \mathbf{case}\ u_1 > n\ \mathbf{of} \\
&\quad true \ : \ nil \\
&\quad false \ : \ cons \ \ (square\ u_1)\ \mathbf{let} u_2 = [\![u_1 + 1]\!]\ \mathbf{in}\ [\![(squares\ (upto\ u_2\ n))]\!] \\
&= \mathbf{case}\ u_1 > n\ \mathbf{of} \\
&\quad true \ : \ nil \\
&\quad false \ : \ cons \ \ (square\ u_1)\ \mathbf{let} u_2 = u_1 + 1\ \mathbf{in}\ h(u_2, n) \\
&= \mathbf{case}\ u_1 > n\ \mathbf{of} \\
&\quad true \ : \ nil \\
&\quad false \ : \ cons \ \ (square\ u_1)\ h(u_1 + 1, n)
\end{aligned}
$$

以上より、

$[\![squares(upto\ u_1\ n)]\!] \ = \ h(u_1, n)$

where

$$
\begin{aligned}
h(u_1, n) \ = \ &\mathbf{case}\ (u_1 > n)\ \mathbf{of} \\
&\quad true \ : \ nil \\
&\quad false \ : \ h(u_1 + 1, n)
\end{aligned}
$$

すると *(1)* は

$= \ \mathbf{let}\ u_1 = 1\ \mathbf{in}\ h(u_1, n)$

$= h(1, n)$ つまり、

$[\![squares(upto\ 1\ n)]\!] \ = \ h(1, n)$ となる。

第 3 章

先行評価言語上での問題点

Wadler の切り払いは、遅延評価言語を対象にして定義されている。従って、先行評価言語上で印付き切り払いを行なうと様々な問題が起こってくる。本章では、最初に先行評価言語上で、印付き切り払いを行なった時に起きる問題点を考察する。次にそのようなことが起きる原因を突き止め、解決法を示す。

3.1 代入に関する問題点（１）

3.1.1 問題点

変換前の関数の引数の集合と変換後の関数の引数の集合が異なる場合に 2 つの関数が等価でない、つまり変換の正当性が保証されない場合が出てくる。

例 3.1.1 次の関数定義のもとでの切り払いを考える。

$f(x) \ = \ \mathbf{case}\ x\ \mathbf{of}$

$\qquad\qquad Nil \ : \ Nil$

$\qquad\qquad Cons(z, zs) \ : \ f(zs)$

$g(x, y) \ = \ Cons(x, y)$ この時、

$[\![f(g(x, y))]\!]$

$= [\![\mathbf{case}\ g(x, y)\ \mathbf{of}$

$\qquad Nil \ : \ Nil$

$\qquad Cons(z, zs) \ : \ f(zs)]\!]$

$= [\![\textbf{case}\ Cons(x,y)\ \textbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z,zs)\ :\ f(zs)]\!]$

$= [\![f(y)]\!]$

$= [\![\textbf{case}\ y\ \textbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z,zs)\ :\ f(zs)]\!]$

$= \textbf{case}\ y\ \textbf{of}$

$\quad Nil\ :\ [\![Nil]\!]$

$\quad Cons(z,zs)\ :\ [\![f(zs)]\!]$

$= \textbf{case}\ y\ \textbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z,zs)\ :\ h(zs)$

ここで、畳み込みを行なうと、

$= h(y)$

よって、$[\![f(g(x,y))]\!]\ =\ h(y)$

$where\ h(y)\ =\ \textbf{case}\ y\ \textbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z,zs)\ :\ h(zs)$

x=*loop*（停止しない項）,y=*cons 2 Nil* とすると、

・変換前

$f(g(loop, cons\ 2\ Nil)) \overset{eager}{\rightarrow} \ldots \overset{eager}{\rightarrow} \ldots$

・変換後

$h(cons\ 2\ Nil) \overset{eager}{\rightarrow} \textbf{case}\ cons\ 2\ Nil\ \textbf{of} \quad \overset{eager}{\rightarrow} h(Nil) \overset{eager}{\rightarrow} \textbf{case}\ Nil\ \textbf{of} \quad \overset{eager}{\rightarrow} Nil$

$\quad Nil\ :\ Nil \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nil\ :\ Nil$

$\quad Cons(z,zs)\ :\ h(zs) \qquad\qquad\qquad\qquad\qquad\qquad Cons(z,zs)\ :\ h(zs)$

変換前と変換後の関数が等価ではない。しかし、遅延評価で見てみると、

変換前と変換後の関数は等価になる。

・変換前

$$f(g(loop, cons\ 2\ Nil)) \overset{lazy}{\rightarrow} \textbf{case}\ g(loop, cons\ 2\ Nil)\ \textbf{of} \overset{lazy}{\rightarrow} \textbf{case}\ cons(loop, cons\ 2\ Nil)\ \textbf{of}$$
$$Nil\ :\ Nil \qquad\qquad Nil\ :\ Nil$$
$$Cons(z, zs)\ :\ f(zs) \qquad\qquad Cons(z, zs)\ :\ f(zs)$$
$$\overset{lazy}{\rightarrow} f(cons\ 2\ Nil) \overset{lazy}{\rightarrow} \textbf{case}\ cons\ 2\ Nil\ \textbf{of} \overset{lazy}{\rightarrow} f(Nil) \overset{lazy}{\rightarrow} \textbf{case}\ Nil\ \textbf{of} \overset{lazy}{\rightarrow} Nil$$
$$Nil\ :\ Nil \qquad\qquad Nil\ :\ Nil$$
$$Cons(z, zs)\ :\ f(zs) \qquad\qquad Cons(z, zs)\ :\ f(zs)$$

・変換後

$$h(cons\ 2\ Nil) \overset{lazy}{\rightarrow} \textbf{case}\ cons\ 2\ Nil\ \textbf{of} \qquad \overset{lazy}{\rightarrow} h(Nil) \overset{lazy}{\rightarrow} \textbf{case}\ Nil\ \textbf{of} \qquad \overset{lazy}{\rightarrow} Nil$$
$$Nil\ :\ Nil \qquad\qquad Nil\ :\ Nil$$
$$Cons(z, zs)\ :\ h(zs) \qquad\qquad Cons(z, zs)\ :\ h(zs)$$

3.1.2 解決策

このように、変換前と変換後の関数にそれぞれ同じ入力を与えたにもかかわらず、出力が異なっている。これは、変換前と変換後の関数の引数の集合が異なることが原因で起こる。変換したい関数がそれ自身再帰関数に変換される場合は変数の消滅はない. 上の例でいえば、変換途中に $T[f(g(v, w))]$ が出て来て再帰関数を定義することによって変換が終了した場合、変数の消滅はない。しかし、上の例ではそのような終り方はしていない。先行評価言語においては、変数の消滅があった場合について、新たな操作を付け加える。
「変換前の変数の集合を引数に持つ関数を新たに定義する。」
この操作によって変数に停止しない項が入った時の等価性が保証される。上の例では次のように新関数を定義する。

$$[\![f(g(x, y))]\!] \ =\ h'(x, y)$$
where $h'(x, y) = h(y)$
$$h(y)\ =\ \textbf{case}\ y\ \textbf{of}$$
$$Nil\ :\ Nil$$
$$Cons(z, zs)\ :\ h(zs)$$

x=loop （停止しない項）,y=cons 2 Nil とすると
$f(g(loop, cons\ 2\ Nil)) \overset{eager}{\rightarrow} \ldots \overset{eager}{\rightarrow}$

$h'(loop, cons\ 2\ Nil) \overset{eager}{\rightarrow} \ldots \overset{eager}{\rightarrow}$
このように、変数の消滅による等価性の問題は解決できる。

3.2 代入に関する問題点（２）

3.2.1 問題点

let 文を使って$\ominus$の項を取り出して変換をしたとき、変換終了後に取り出した項を戻す。しかし、戻す位置によっては、等価性が成り立たなくなってしまう場合がある。

例 3.2.1 次の関数定義のもとでの切り払いを考える。

$$
\begin{aligned}
f(x,y,z) \ &= \ \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad true \ : \ y \\
&\qquad\qquad false \ : \ z \\
g(x) \ &= \ \textbf{case}\ x > 0\ \textbf{of} \\
&\qquad\qquad true \ : \ g(x+1) \\
&\qquad\qquad false \ : \ 0 \\
h(x,y) \ &= \ f(x,y,g(2))
\end{aligned}
$$

今、$h(x,y)$ という関数を切り払う。

$[\![h(x,y)]\!]$
$= [\![f(x,y,g(2))]\!]$
$= [\![\textbf{let}\ v = g(2)\ \textbf{in} f(x,y,v)]\!]$
$= \textbf{let}\ v = [\![g(2)]\!]\ \textbf{in}[\![f(x,y,v)]\!] \qquad --- (a)$

$[\![g(2)]\!]$
$= ldefo\textbf{let}\ w = 2\ \textbf{in} g(w)]\!]$
$= \textbf{let}\ w = [\![2]\!]\ \textbf{in}[\![g(w)]\!]$
$= \textbf{let}\ w = 2\textbf{in}[\![g(w)]\!] \qquad --- (b)$

ここで、
$[\![g(w)]\!]$

$$
\begin{aligned}
&= [\![case\ w > 0\ \mathbf{of} \\
&\qquad true\ :\ g(w+1) \\
&\qquad false\ :\ 0]\!] \\
&= case\ w > 0\ \mathbf{of} \\
&\qquad true\ :\ [\![g(w+1)]\!] \\
&\qquad false\ :\ [\![0]\!] \\
&= case\ w > 0\ \mathbf{of} \\
&\qquad true\ :\ [\![\mathbf{let}\ w' = w+1\ \mathbf{in}\ g(w')]\!] \\
&\qquad false\ :\ 0 \\
&= case\ w > 0\ \mathbf{of} \\
&\qquad true\ :\ \mathbf{let}\ w' = [\![w+1]\!]\ \mathbf{in}\ [\![g(w')]\!] \\
&\qquad false\ :\ 0 \\
&= case\ w > 0\ \mathbf{of} \\
&\qquad true\ :\ \mathbf{let}\ w' = [\![w]\!] + [\![1]\!]\ \mathbf{in}\ [\![g'(w')]\!] \\
&\qquad false\ :\ 0 \\
&= case\ w > 0\ \mathbf{of} \\
&\qquad true\ :\ \mathbf{let}\ w' = w+1\ \mathbf{in}\ g'(w') \\
&\qquad false\ :\ 0 \\
&= case\ w > 0\ \mathbf{of} \\
&\qquad true\ :\ g'(w+1) \\
&\qquad false\ :\ 0
\end{aligned}
$$

ここで畳み込む。$[\![g(w)]\!] = g'(w)$

$$
\begin{aligned}
where\ g'(w)\ &=\ case\ w > 0\ \mathbf{of} \\
&\qquad\qquad true\ :\ [\![g'(w+1)]\!] \\
&\qquad\qquad false\ :\ 0
\end{aligned}
$$

(b) は

$= \mathbf{let}\ w = 2\ \mathbf{in}\ g'(w)$

$= g'(2)$

よって、$[\![g(2)]\!]\ =\ g'(2)$

$[\![f(x,y,w)]\!]$

$= [\![\textbf{case } x \textbf{ of}$

$\quad true \ : \ y$

$\quad false \ : \ w]\!]$

$= \textbf{case } x \textbf{ of}$

$\quad true \ : \ [\![y]\!]$

$\quad false \ : \ [\![w]\!]$

$= \textbf{case } x \textbf{ of}$

$\quad true \ : \ y$

$\quad false \ : \ w$

ここで、(a) は、

$= \textbf{let } v = g'(2) \textbf{ in case } x \textbf{ of}$

$\quad true \ : \ y$

$\quad false \ : \ w$

$= \textbf{case } x \textbf{ of}$

$\quad true \ : \ y$

$\quad false \ : \ g'(2)$

以上より、$[\![h(x,y)]\!] = h'(x,y)$

$where \ h'(x,y) = \textbf{case } x \textbf{ of}$

$\quad true \ : \ y$

$\quad false \ : \ g'(2)$

$x = true, y = 3$ とする。この時、

$h(true,3) \overset{eager}{\rightarrow} f(true,3,g(2)) \overset{eager}{\rightarrow} \ldots$

よって、$h(true,3)$ は停止しない。しかし、変換後の関数は停止する。

$h'(true,3) \overset{eager}{\rightarrow} \textbf{case } true \textbf{ of} \overset{eager}{\rightarrow} 3$

$\quad true \ : \ 3$

$\quad false \ : \ g(2)$

よって、変換前と変換後の関数は等価ではない。

遅延評価で *2* つの関数を見てみると変換前と変換後の関数は等価である。

$$h(true,3) \overset{lazy}{\rightarrow} f(true,3,g(2)) \overset{lazy}{\rightarrow} \textbf{case}\ true\ \textbf{of} \overset{eager}{\rightarrow} 3$$
$$true\ :\ 3$$
$$false\ :\ g(2)$$

$$h'(true,3) \overset{lazy}{\rightarrow} \textbf{case}\ true\ \textbf{of} \overset{lazy}{\rightarrow} 3$$
$$true\ :\ 3$$
$$false\ :\ g(2)$$

3.2.2 解決策

原因は **let** 式を戻す時に戻す項が必ずしも評価されない可能性のある位置に来たことである。問題があるのは *(10,11)* のルールで取り出した項に (6) が含まれている場合である。(6) が含まれている項を戻す時に、戻すところが **case** 式の分岐変数以外のところにある場合に、問題が起こる可能性がある。そのような場合は、変換が終了後に次の操作を行なうことによって解決できる。

「取り出した項を関数の初期値として持つような関数に変換する。」

例 *3.2.1* で **let** 式を戻すところを変更する。

$$= \textbf{let}\ v = g'(2)\ \textbf{in case}\ x\ \textbf{of}$$
$$true\ :\ y$$
$$false\ :\ w$$
$$= ff(x,y,g'(2))$$
$$where\ ff(x,y,w)\ = \textbf{case}\ x\ \textbf{of}$$
$$true\ :\ y$$
$$false\ :\ w$$

$[\![h(x,y)]\!]\ =\ ff(x,y,g'(2))$

とすれば、

$x = true, y = 3$（停止しない項）の時、等価になる。

$ff(x,y,g'(2)) \overset{eager}{\rightarrow} \ldots$　（$g'(2)$ で $loop$）

3.3 展開に関する問題点（１）

3.3.1 問題点

3.1 節では変数が消滅する時の問題点について考察してきたが、この節では全ての変数が消滅しない時の変換の正当性について考察する。Wadler の切り払いでは全ての変数に値が入った時でも、等価性が成り立たない場合がある。

例 3.3.1 次の関数定義のもとでの切り払いを考える。

$$
\begin{aligned}
f(x,y,z) \;=\; & \textbf{case}\ x\ \textbf{of}\\
& \quad true \;:\; y\\
& \quad false \;:\; cons\ z\ nil\\
g(x,y) \;=\; & \textbf{case}\ x\ \textbf{of}\\
& \quad true \;:\; cons\ y\ nil\\
& \quad false \;:\; g(false, y)
\end{aligned}
$$

今、$f(x, g(y,z), w)$ という関数の切り払いを考える。

$$
\begin{aligned}
& [\![f(x, g(y,z), w)]\!]\\
=\; & [\![\textbf{case}\ x\ \textbf{of}\\
& \quad true \;:\; g(y,z)\\
& \quad false \;:\; cons\ w\ nil]\!]\\
=\; & \textbf{case}\ x\ \textbf{of}\\
& \quad true \;:\; [\![g(y,z)]\!]\\
& \quad false \;:\; [\![cons\ w\ nil]\!]\\
=\; & \textbf{case}\ x\ \textbf{of}\\
& \quad true \;:\; [\![\textbf{case}\ y\ \textbf{of}\\
& \qquad\qquad true \;:\; cons\ z\ nil\\
& \qquad\qquad false \;:\; g(false, z)]\!]\\
& \quad false \;:\; cons\ [\![w]\!]\ [\![nil]\!]\\
=\; & \textbf{case}\ x\ \textbf{of}
\end{aligned}
$$

$$
\begin{array}{l}
\qquad true \ : \ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad true \ : \ [\![cons\ z\ nil]\!] \\
\qquad\qquad\qquad false \ : \ [\![g(false, z)]\!] \\
\qquad false \ : \ cons\ w\ nil \\
= \ \textbf{case}\ x\ \textbf{of} \\
\qquad true \ : \ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad true \ : \ cons\ [\![z]\!]\ [\![nil]\!] \\
\qquad\qquad\qquad false \ : \ [\![\textbf{let}\ u_1 = false\ \textbf{in}\ g(u_1, z)]\!] \\
\qquad false \ : \ cons\ w\ nil \\
= \ \textbf{case}\ x\ \textbf{of} \\
\qquad true \ : \ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad true \ : \ cons\ z\ nil \\
\qquad\qquad\qquad false \ : \ \textbf{let}\ u_1 \ = \ [\![false]\!]\ \textbf{in}\ [\![g(u_1, z)]\!] \\
\qquad false \ : \ cons\ w\ nil \\
= \ \textbf{case}\ x\ \textbf{of} \\
\qquad true \ : \ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad true \ : \ cons\ z\ nil \\
\qquad\qquad\qquad false \ : \ \textbf{let}\ u_1 \ = \ false\ \textbf{in}\ g'(u_1, z) \\
\qquad false \ : \ cons\ w\ nil \\
= \ \textbf{case}\ x\ \textbf{of} \\
\qquad true \ : \ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad true \ : \ cons\ z\ nil \\
\qquad\qquad\qquad false \ : \ g'(false, z) \\
\qquad false \ : \ cons\ w\ nil \\
= \ \textbf{case}\ x\ \textbf{of} \\
\qquad true \ : \ g'(y, z) \\
\qquad false \ : \ cons\ w\ nil
\end{array}
$$

以上より、$[\![f(x, g(y, z), w)]\!] \ = \ h(x, y, z, w)$

$where\ h(x, y, z, w) \ = \ \textbf{case}\ x\ \textbf{of}$

$$
\qquad\qquad\qquad\qquad true \ : \ g'(y, z)
$$

$$false \ : \ cons\ w\ nil$$

$$g'(x,y) \ = \ \textbf{case}\ x\ \textbf{of}$$
$$true \ : \ cons\ y\ nil$$
$$false \ : \ g'(false, y)$$

いま、*x=false,y=false,z=false,w=false* とすると、先行評価で評価した場合、変換前と変換後の関数は等価ではない。

$f(false, g(falsefalse,), false) \overset{eager}{\rightarrow} \ldots (g(false, false)$ で $loop)$

$$h(false, false, false, false,) \overset{eager}{\rightarrow} \textbf{case}\ false\ \textbf{of} \overset{eager}{\rightarrow} cons\ false\ nil$$
$$true \ : \ g'(false, false)$$
$$false \ : \ cons\ false\ nil$$

遅延評価で見た場合は等価である。

$$f(false, g(falsefalse,), false) \overset{lazy}{\rightarrow} \textbf{case}\ false\ \textbf{of} \overset{lazy}{\rightarrow} cons\ false\ nil$$
$$true \ : \ g'(false, false)$$
$$false \ : \ cons\ false\ nil$$
$$h(false, false, false, false,) \overset{lazy}{\rightarrow} \textbf{case}\ false\ \textbf{of} \overset{lazy}{\rightarrow} cons\ false\ nil$$
$$true \ : \ g'(false, false)$$
$$false \ : \ cons\ false\ nil$$

3.3.2 解決策

これは、$g(x,y)$ が x=false のときに、値が定義されていない部分関数であることが原因で起こる。遅延評価では、$g(x,y)$ の評価が必要になるまで評価されないが、先行評価では引数は全て展開する前に評価するので、変数だけを最初の引数としてとる場合と動きが異なってくる。今回は、この問題に関しては次のような制限を加えることによって解決した。

「$\oplus$に印付けられた関数は全関数 (total funtion) のみを取り扱う。」

部分関数を取り扱っても切り払いはできる。しかし、その時は正当性が成り立たなくなる。

3.4 展開に関する問題点（２）

3.4.1 問題点

関数の定義の右辺に 2 回以上同じ変数が現れる関数を含む合成関数の切り払いを行なった時に効率が悪くなる場合がある。

例 3.4.1 次の関数定義のもとでの切り払いを考える。

$f(x) \ = \ cons\ x\ cons\ x\ nil$
flip—木の右の要素と左の要素を入れ換える
$flip(xt) \ = \ \mathbf{case}\ xt\ \mathbf{of}$
$\qquad\qquad\qquad Leaf\ z \ : \ Leaf\ z$
$\qquad\qquad\qquad Branch(yt, zt) \ : \ Branch(flip(zt), flip(yt))$

今、$f(flip(xt))$ という関数の切り払いを考える。
変換の途中で $[\![(flip(xt)]\!]$ という変換が *2* 回出てくるので先に変換過程を示す。

$[\![flip(xt)]\!]$
$= [\![\mathbf{case}\ xt\ \mathbf{of}$
$\qquad leaf\ z \ : \ leaf\ z$
$\qquad branch(yt, zt) \ : \ branch(flip(zt), flip(yt))]\!]$
$= \mathbf{case}\ xt\ \mathbf{of}$
$\qquad leaf\ z \ : \ [\![leaf\ z]\!]$
$\qquad branch(yt, zt) \ : \ [\![branch(flip(zt), flip(yt))]\!]$
$= \mathbf{case}\ xt\ \mathbf{of}$
$\qquad leaf\ z \ : \ leaf\ [\![z]\!]$
$\qquad branch(yt, zt) \ : \ branch([\![flip(zt)]\!], [\![flip(yt)]\!])$
$= \mathbf{case}\ xt\ \mathbf{of}$
$\qquad leaf\ z \ : \ leaf\ z$
$\qquad branch(yt, zt) \ : \ branch(h(zt), h(yt))$
以上より、$[\![flip(xt)]\!] = h(xt)$
$where\ h(xt) \ = \ \mathbf{case}\ xt\ \mathbf{of}$
$\qquad\qquad\qquad Leaf\ z \ : \ Leaf\ z$

$$branch(yt, zt) \ : \ Branch(h(zt), h(yt))$$

$[\![f(flip(xt))]\!]$

$= [\![cons \ flip(xt) \ cons \ flip(xt) \ nil]\!]$

$= cons \ [\![flip(xt)]\!] \ [\![cons \ flip(xt) \ nil]\!]$

$= cons \ h(xt) \ cons \ [\![flip(xt)]\!] \ [\![nil]\!]$

$= cons \ h(xt) \ cons \ h(xt) \ nil$

以上より、$[\![f(flip(xt))]\!] \ = h'(xt)$

$where \ h'(xt) \ = \ cons \ h(xt) \ cons \ h(xt) \ nil$

$h(xt) \ = \ \mathbf{case} \ xt \ \mathbf{of}$

$Leaf \ z \ : \ Leaf \ z$

$Branch(yt, zt) \ : \ Branch(h(zt), h(yt))$

$xt = Branch(Branch(Leaf1, Leaf2), Branch(Leaf3, Leaf4))$ とする。

$flip(xt) \overset{eager*14}{\rightarrow} Branch(Branch(Leaf4, Leaf3), Branch(Leaf2, Leaf1))$

$yt = Branch(Branch(Leaf4, Leaf3), Branch(Leaf2, Leaf1))$ とする。

この時、先行評価、遅延評価共に $flip(xt)$ を評価すると *14* ステップで値 yt にたどり着く。ここで、変換前の関数 $f(flip(xt))$ と変換後の関数 $h(xt)$ の簡約のステップ数についてみてみる。

・変換前

$f(flip(xt)) \ \overset{eager*14}{\rightarrow} \ f(yt) \ \overset{eager}{\rightarrow} \ cons \ yt \ cons \ yt \ nil$ (15 ステップ)

・変換後

$h'(xt) \ \overset{eager}{\rightarrow} \ cons \ h'(xt) \ cons \ h'(xt) \ nil \ \overset{eager*28}{\rightarrow} \ cons \ yt \ cons \ yt \ nil$ (29 ステップ)

先行評価においては変換の途中で $flip(xt)$ が複製されたのが原因で効率が悪くなった。遅延評価でのステップ数についてみてみる。遅延評価の場合はステップ数が悪くなることはない。

・変換前

$f(flip(xt)) \ \overset{lazy}{\rightarrow} \ cons \ flip(xt) \ cons \ flip(xt) \ nil \ \overset{lazy*28}{\rightarrow} \ cons \ yt \ cons \ yt \ nil$ (29 ステップ)

・変換後

$h'(xt) \ \overset{lazy}{\rightarrow} \ cons \ h(xt) \ cons \ h(xt) \ nil \ \overset{lazy*28}{\rightarrow} \ cons \ yt \ cons \ yt \ nil$ (29 ステップ)

3.4.2 解決策

遅延評価言語では、展開に関して改善関係が成り立つ。しかし、先行評価言語では改善関係が成り立たない場合がある。上の例では flip(xt) が右辺でコピーされるためにステップ数が増える。この問題点は次のようにして解決する。

「展開によって計算が複製される項を **let** 式で取り出し、展開の規則を適用する。
そして変換後に、新関数の引数として **let** 式で取り出された部分を取るようにする。」

このような操作を行なえば、問題となる項は、必ず複製される前に評価されることになる。そうすれば、ステップ数に関して効率が悪くなることはなくなる。上の例で改善策を考えてみる。

$h'(xt) \ = \ cons\ h(xt)\ cons\ h(xt)\ nil$

h(xt) が 2 度出現しているので次のような関数にできれば効率は悪くならない。

$g(h'(xt))$

where $g(x) \ = \ cons\ x\ cons\ x\ nil$

第 4 章

先行評価向け印付き切り払い

本章では、3 章で示した解決策をもとに、印付き切り払いを先行評価言語向けに再構築する。そして、先行評価言語向け印付き切り払いで停止性、正当性が成り立つことを示す。

4.1 項・関数定義・再帰的データ型

定義 4.1.1 (項) 取り扱う項は、定義 *2.1.1* と同じである。しかし、中間データとして不都合でない数値型、ブール型の項を区別する。

$$
\begin{array}{rcll}
t & ::= & vv & \\
 & | & (c(t_1,\ldots,t_k))^{\oplus} & (1)\\
 & | & (f(t_1,\ldots,t_k))^{\oplus} & (2)\\
 & | & (\mathbf{case}\ t_0\ \mathbf{of}\ p_1\ :\ t_1\ |\ldots|p_n\ :\ t_n)^{\oplus} & (3)\\
vv & ::= & v & (4)\\
 & | & c^{\ominus} & (5)\\
 & | & (f(t_1,\ldots,t_k))^{\ominus} & (6)\\
 & | & (\mathbf{case}\ t_0\ \mathbf{of}\ p_1\ :\ t_1\ |\ldots|p_n\ :\ t_n)^{\ominus} & (7)
\end{array}
$$

定義 4.1.2 (関数定義) 関数定義は、定義 *2.3.1* と次の点でことなる。*1* つ目は、右辺に出現する変数が線形でなくても良い。*2* つ目は、関数定義の右辺において構成子式の中に **case** 式が入ってはいけない。*3* つ目は、構成子式において、基底データの部分に構成子式が入ってはいけない。*4* つ目は、関数定義の右辺が **case** 式の場合、分岐以降で再帰的

データを表す変数 *(xs)* のところにかかる関数は次のような形をしていないといけない。
関数定義の右辺が xs である。
もしくは **case** 式の分岐変数のところに xs が来る。

定義 4.1.3 (再帰関数の定義) 再帰関数は **case** 式によってのみ定義されているが、再帰の部分において$\oplus$に印付けられた項を表す変数の位置は変わってはいけない。

$$
\begin{aligned}
f(x,y,z) \ = \ & \textbf{case}\ x\ \textbf{of} \\
& \quad Nil \ : \ \ldots \\
& \quad Cons(w,ws) \ : \ \ldots f(zs,y,z)
\end{aligned}
$$

このように、y と z の位置はそのままでなければいけない。

例 4.1.1 (線形でない例) *Wadler* 方式で取り扱えた項に加えて次のような項を取り扱えるようになった。ただし、変数は全てリスト、もしくは木の変数とする。

$$
\begin{aligned}
& cons\ x\ cons\ x\ nil \\
& Branch(Branch(Leaf\ z, Leaf\ z), Branch(Leaf\ z, Leaf\ z) \\
& f(x) \ = \ g(x,x) \\
& f(x,y) \ = \ \textbf{case}\ x\ \textbf{of} \\
& \quad nil \ : \ cons\ y\ cons\ y\ nil \\
& \quad cons(z,zs) \ : \ cons\ z\ f(zs,y) \\
& f(x,y) \ = \ \textbf{case}\ x\ \ \textbf{of} \\
& \quad nil \ : \ cons\ y\ cons\ y\ nil \\
& \quad cons(z,zs) \ : \ cons\ z\ cons\ z\ f(zs,y) \\
& f(x,y) \ = \ \textbf{case}\ x\ \ \textbf{of} \\
& \quad Leaf\ z \ : \ Branch(Leaf\ z, Leaf\ y) \\
& \quad Branch(xt,yt) \ : \ Branch(f(xt,y), f(yt,y))
\end{aligned}
$$

定義 4.1.4 (絶対評価位置) 項 t の中において x が次の位置にある時、x は絶対位置にあるという。

- t=x
- 構成子式の中で絶対評価位置にある。

- 関数式の中で絶対評価位置にある。
- **case** 式の分岐項の位置にありかつ分岐項の中で絶対評価位置にある。

定義 4.1.5 (切り払いの対象となる項) 切り払いの対象となる項は関数の組み合わさった形をしている。そして、初期値として$\ominus$に印付けられた項の出現を許す。ただし、$\ominus$の項の内、**case** 式は除く。これは、変数は、変換開始時に絶対評価位置にあることを前提に切り払いを行なうためである。

4.2 変換規則を適用する前に

変換したい関数を切り払う前に行なう処理について説明する。Wadler の印付切り払いで は、展開の規則 (3), (6) の規則を使う時に数値型やブール型の項を let 文で取り出していた。例えばつきの関数を切り払う。

$f(3, x, y)$

この時、変換規則を適用する前に$\ominus$に印付けられたものを let 文で取り出してから変換していた。つまり、

$let\ v_1 = 3\ in\ f(v_1, x, y)$

としてから変換規則を適用していた。今回はさらに、関数を展開する前に$\oplus$ に印付けられた項で線形でない変数の位置にある項も **let** 式で変数に置き換えて展開する。

例 4.2.1 ($\oplus$の項の変数への置き換え) 次の関数を定義する。

$$
\begin{aligned}
f &: int\ list \rightarrow (int\ list)\ list \\
f(x) &= cons\ x\ cons\ x\ nil \\
append &: (int\ list * int\ list) \rightarrow int\ list \\
append(xs, ys) &= \textbf{case}\ xs\ \textbf{of} \\
&\qquad Nil : ys \\
&\qquad Cons(z, zs) : Cons(z, append(zs, ys))
\end{aligned}
$$

今、$f(append(xs, ys))$ という関数の切り払いを考える。まず、最初の変換として関数 fを展開するのだが、この時、そのまま展開すると *append(xs,ys)* が *2* 回出現することになる。そのまま展開すると、*3* 章で述べたように効率が悪くなるので *append(xs,ys)* は関数の初

期値としてとっておく。そのため、*append(xs,ys)* を w と置き換えて切り払いを行なう。つまり $T[\mathbf{let}^{\oplus}\ w\ =\ append(xs, ys)\ \mathbf{in}\ f(w)]$ の切り払いを行なう。

4.3 変換規則の適用

3 章で述べた問題点を解決するために定義 2.1.4 で定義した変換規則の適用の方法を変更する。先行評価においては変換規則を適用すると効率が悪くなったり、プログラムの意味が変わったりする場合が出てくる。そのような状況においては切り払いをしない方がいいので変換規則を適用しないようにする。

4.3.1 変換規則

定義 4.3.1 (変換規則) W を複数回現れる項を表す変数の集合とする。変換規則は次のように定義される。

1. (a) $[\![v]\!]\ =\ v$

   (b) $[\![v]\!]\ =\ v_i \qquad\qquad v \in W$

2. $[\![c(t_1, \ldots, t_k)]\!]\ =\ c([\![t_1]\!], \ldots, [\![t_k]\!])$

3. (a) $[\![f(t_1, \ldots, t_k)]\!]\ =\ [\![t[t_1/x_1, \ldots, t_k/x_k]]\!]$
   where f is defined by $f(x_1, \ldots, x_k) = t$

   (b) $[\![f(x)]\!]\ =\ x_i \qquad\qquad v \in W$

4. $[\![\mathbf{case}\ v\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!] = \mathbf{case}\ v\ \mathbf{of}\ p'_1 : [\![t'_1]\!] | \ldots | p'_n : [\![t'_n]\!]$

5. $[\![\mathbf{case}\ c(t_1, \ldots, t_k)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!] = [\![t'_i[t_1/v_1, \ldots, t_k/v_k]]\!]$
   $where\ p'_i\ =\ c(v_1, \ldots, v_k)$

6. (a) $[\![\mathbf{case}\ f(t_1, \ldots, t_k)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$
   $= [\![\mathbf{case}\ t[t_1/v_1, \ldots, t_k/v_k]\ of\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$
   where f is defined by $f(v_1, \ldots, v_k) = t$

   (b) $[\![\mathbf{case}\ f(x)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$
   $= [\![\mathbf{case}\ x_i\ of\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!] \qquad\qquad v \in W$

7. $[\![\mathbf{case}\ (\mathbf{case}\ t\ \mathbf{of}\ p_1 : t_1 | \ldots | p_m : t_m)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$
   $= [\![\ \mathbf{case}\ t\ \mathbf{of}\ p_1\ :\ (\mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)$
   $p_m\ :\ (\mathbf{case}\ t_m\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)]\!]$

8. $[\![f(t_1^{\ominus}, \ldots, t_k^{\ominus})^{\ominus}]\!] = f([\![t_1]\!], \ldots, [\![t_k]\!])$
   （ただし、f が原始関数の場合は $t_1, \ldots, t_k$ が$\oplus$でも適用する。）

9. $[\![\mathbf{case}\ f(t_1^{\ominus}, \ldots, t_k^{\ominus})^{\ominus}\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$
   $= \mathbf{case}\ f([\![t_1]\!], \ldots, [\![t_k]\!])\ \mathbf{of}\ p'_1 : [\![t'_1]\!] | \ldots | p'_n : [\![t'_n]\!]$
   （ただし、f が原始関数の場合は $t_1, \ldots, t_k$ が$\oplus$でも適用する。）

10. $[\![\mathbf{let}^{\ominus}\ v^{\ominus} = t_0^{\ominus}\ \mathbf{in}\ t_1]\!]$
    $= \mathbf{let}^{\ominus}\ v^{\ominus} = [\![t_0^{\ominus}]\!]\ \mathbf{in}\ [\![t_1]\!]$

11. $[\![\mathbf{case}\ (\mathbf{let}^{\ominus}\ v^{\ominus} = t_0^{\ominus}\ \mathbf{in}\ t_1)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$
    $= \mathbf{let}^{\ominus}\ v^{\ominus} = [\![t_0^{\ominus}]\!]\ \mathbf{in}\ [\![\mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$

12. $[\![\mathbf{let}^{\oplus}\ v^{\oplus} = t_0^{\oplus}\ \mathbf{in}\ t_1]\!]$
    $= \mathbf{let}^{\oplus}\ v^{\oplus} = [\![t_0^{\oplus}]\!]\ \mathbf{in}\ [\![t_1]\!]$

13. $[\![\mathbf{case}\ (\mathbf{let}^{\oplus}\ v^{\oplus} = t_0^{\oplus}\ \mathbf{in}\ t_1)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$
    $= \mathbf{let}^{\oplus}\ v^{\oplus} = [\![t_0^{\oplus}]\!]\ \mathbf{in}\ [\![\mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n]\!]$

変更点のあったところの説明をする。(2), (4), (7), (10), (11) の 5 つの規則に関しては、変更点はない。

- (1) の規則
  変数が複数回出現するものでない場合は、そのままにする。複数回出現する場合は、引数として何をとるかを決めるための操作を行なう。その操作については次の節で述べる。

- (3) の規則
  $[\![f(t_1, \ldots, t_k)]\!]$ を適用する時は次のような場合が考えられる。

  1. すべてが 1 引数の関数で、複数回出現する変数だけが存在する場合、展開を行なわないで初期値として何をとるのかを決める操作を行なう。

2. それ以外の場合は展開する。

例 4.3.1 $x,y,$ を右辺での出現が線形である変数とし、z を線形ではない変数とする。

T[f(x,y)]---（3）の規則を適用して展開する。
T[f(z,z)]---（3）の規則を適用して展開する。
T[g(z)]---展開しないで初期値として何をとるのかを決める操作を行なう。

- (5) の規則
(5) の規則はセレクタの部分がある一つのパターンとマッチした時に v_1 から v_k にそれぞれ当てはまる値を代入するものである。図 4.1 では、この規則を適用してもいい場合と、いけない場合の場合分けがしてある。適用してはいけない場合は、正当性が成り立たないような変換が行なわれる可能性がある。このような場合は、切り払いを中止する。

例 4.3.2 次のような状況を考える。x は線形とする。

```
case (cons f(x) nil) of
     nil:nil
     cons(z,zs):cons g(z) cons g(z) h(zs)
```

この項を（5）の規則を使って変換すると次のようになる。

```
cons g(f(x)) cons g(f(x)) h(nil)
```

$g(x) = x$（何もしない関数）とする。f が非常に計算のかかる関数だとすると変換前の式で f を計算させてから g の関数を計算させた方がステップ数が少ない場合が出てくる。このような場合は変換規則を適用しない。

- (6) の規則
（6）の変換規則を適用する場合、（3）と同じような場合が考えられる。

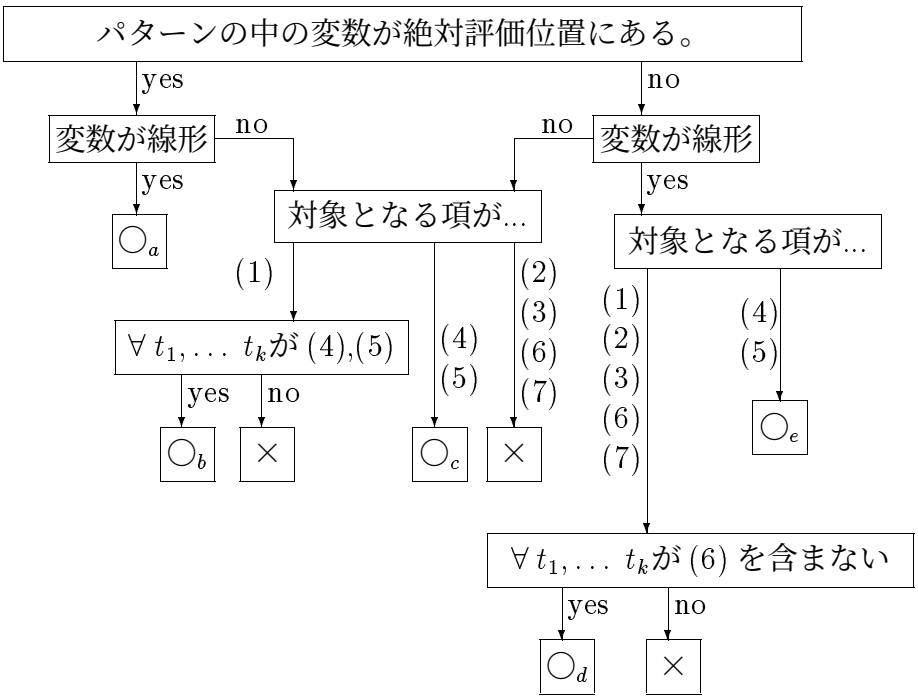

図 4.1: (5) の規則の適用条件

1. すべてが 1 引数の関数で、複数回出現する変数だけが存在する場合、展開を行なわないで初期値として何をとるのかを決める操作を行なう。
2. それ以外の場合は展開する。

- (8), (9) の規則
  原始関数は展開のしようがないので、引数に$\oplus$の項が含まれている場合でも適用する。
- (12), (13) の規則
  (10), (11) 同様の変換を行なう。

4.3.2 変換の停止性

この節では変換の手続きが停止することを証明する。そのために必要な定義を行なう。

定義 4.3.2 (項の深さ) 項の深さを次のように定義する。

$$
\begin{aligned}
N[\![v]\!] &= 0 \\
N[\![c]\!] &= 0 \\
N[\![c(t_1,\ldots,t_n)]\!] &= max\{N[\![t_1]\!],\ldots,N[\![t_n]\!]\} \\
N[\![f(t_1,\ldots,t_n)]\!] &= 1 + max\{N[\![t_1]\!],\ldots,N[\![t_n]\!]\} \\
N[\![\mathbf{case}\ t_0\ \mathbf{of}\ p_1\ :\ t_1\ |\ldots|p_n\ :\ t_n]\!] &= N[\![t_0]\!]\ +\ max\{N[\![t_1]\!],\ldots,N[\![t_n]\!]\} \\
N[\![\mathbf{let} v\ =\ t_1\ \mathbf{in}\ t_2]\!] &= max\{N[\![t_1]\!], N[\![t_2]\!]\}
\end{aligned}
$$

今、次のことが成立する。

補題 4.3.1 (項の深さと変換規則の関係) 項の深さが n の関数を切り払う場合、n より大きい深さを持つ関数の展開は起こらない。また、関数の展開の規則以外の変換規則を適用する時には、項の深さが深くはならない。

(証明) 項 t の構造に関する帰納法で示す。

＜場合 1 (変数)：$t = v$＞ 変換規則 (1) で変換するのだが、変換前後の項の深さはともに 0 である。よって成り立つ。

＜場合 2 (構成子式)：$t = c(t_1,\ldots,t_k)$ ＞ 部分項 $t_1,\ldots,t_k$に関して $N[\![T[\![t_i]\!]]\!] \leq N[\![t_i]\!]$ と仮定する。構成子式は変換規則 (2) で次のように変換される。

$[\![c(t_1,\ldots,t_k)]\!] \;=\; c([\![t_1]\!],\ldots,[\![t_k]\!])$

項の深さの定義より変換前の項の深さは

$N[\![c(t_1,\ldots,t_n)]\!] \;=\; max\{N[\![t_1]\!],\ldots,N[\![t_n]\!]\}$

変換後の項の深さは

$N[\![c([\![t_1]\!],\ldots,[\![t_k]\!])]\!] \;=\; max\{N[\![T[\![t_1]\!]]\!],\ldots,N[\![T[\![t_n]\!]]\!]\}$

仮定より、$N[\![c([\![t_1]\!],\ldots,[\![t_k]\!])]\!] \leq N[\![c(t_1,\ldots,t_n)]\!]$ となる。

＜場合 3 (関数式)：$t = f(t_1,\ldots,t_k)$ **＞** 次の場合が考えられる。

＜場合 3a：関数が$\ominus$に印付けられている＞ 部分項 $t_1,\ldots,t_k$に関して $N[\![T[\![t_i]\!]]\!] \leq N[\![t_i]\!]$ と仮定する。$f(t_1,\ldots,t_k)^{\ominus}$は変換規則 (8) より $f([\![t_1]\!],\ldots,[\![t_k]\!])^{\ominus}$に変換される。変換前の項の深さは $1+max\{N[\![t_1]\!],\ldots,N[\![t_n]\!]\}$ である。変換後の項の深さは $1+max\{N[\![T[\![t_1]\!]]\!],\ldots,N[\![T[\![t_n]\!]]\!]\}$ である。仮定より $N[\![T[\![t_i]\!]]\!] \leq N[\![t_i]\!]$ なので $N[\![T[\![f(t_1,\ldots,t_k)]\!]]\!] \leq N[\![f(t_1,\ldots,t_k)]\!]$ となる。

＜場合 3b：変数が複数回出現するもののみである＞ この場合、関数はひとまず変数に置き換えられて変換が終了するので変換後の項の深さは 0 である。よって $N[\![T[\![f(t_1,\ldots,t_k)]\!]]\!] \leq N[\![f(t_1,\ldots,t_k)]\!]$ がいえる。

＜場合 3c：関数を展開する＞ 関数を展開する前の項の深さは、
$1+max\{N[\![t_1]\!],\ldots,N[\![t_n]\!]\}$。右辺の t の項の深さは 1 以上になることがあるので $N[\![t[t_1/v_1,\ldots,t_n/v_n]]\!] \geq N[\![f(t_1,\ldots,t_k)]\!]$ となることがある。しかし、深くなった部分に規則を適用する時は **let** 式を用いて $1+max\{N[\![t_1]\!],\ldots,N[\![t_n]\!]\}$ まで分割される。したがって変換前の項の深さ以上の深さを持つ項を展開することはない。

＜場合 3d：前に出現した関数なので畳み込む＞ 新関数は $fun(x_1,\ldots,x_k)$ の形をしているので深さは 1 になる。変換前の関数の深さは 1 以上なので
$N[\![T[\![f(t_1,\ldots,t_k)]\!]]\!] \leq N[\![f(t_1,\ldots,t_k)]\!]$ がいえる。

＜場合 4 (case 式)：$t = \mathbf{case}\ t_0\ \mathbf{of}\ p_1\ :\ t_1\,|\ldots|\,p_n\ :\ t_n$**＞** 分岐変数の位置にある項の場合分けにより、次の場合が考えられる。

＜分岐項が変数（$t_0 = v$）の場合＞ 部分項 $t_1,\ldots,t_k$に関して $N[\![T[\![t_i]\!]]\!] \leq N[\![t_i]\!]$ と仮定する。(4) の規則によって変換される。変換前の項の深さは

$max\{N[\![t_1]\!], \ldots, N[\![t_n]\!]\}$。変換後の項の深さは $max\{N[\![[\![t_1]\!]]\!], \ldots, N[\![[\![t_n]\!]]\!]\}$。仮定より、変換で項の深さは増加しない。

＜分岐項が構成子式（$t_0 = c(t'_1, \ldots, t'_k)$）の場合＞ (5) の規則によって変換される。変換前の項の深さは $max\{N[\![t'_1]\!], \ldots, N[\![t'_k]\!]\} + max\{N[\![t_1]\!], \ldots, N[\![t_k]\!]\}$。変換後の項の深さは、$max\{N[\![t'_1]\!], \ldots, N[\![t'_k]\!]\} + N[\![t_i]\!]$ になる。よって、変換後に項の深さは増加しない。

＜分岐項が関数式（$t_0 = f(t'_1, \ldots, t'_k)$）の場合＞ (6) の規則によって変換される。変換前の項の深さは $1 + max\{N[\![t'_1]\!], \ldots, N[\![t'_k]\!]\} + max\{N[\![t_1]\!], \ldots, N[\![t_k]\!]\}$。関数の展開によって項の深さが増加する場合もあるが、深くなった部分は **let** 式を用いて $1 + max\{N[\![t'_1]\!], \ldots, N[\![t'_k]\!]\} + max\{N[\![t_1]\!], \ldots, N[\![t_k]\!]\}$ まで分割される。したがって変換前の項の深さ以上の深さを持つ項を展開することはない。

＜分岐項が case 式 $t = \mathbf{case}\ t_0\ \mathbf{of}\ p'_1\ :\ t'_1 \mid \ldots \mid p'_n\ :\ t'_n$ の場合＞ (7) の規則によって変換される。変換前の項の深さは
$N[\![t_0]\!] + max\{N[\![t'_1]\!], \ldots, N[\![t'_n]\!]\} + max\{N[\![t'_1]\!], \ldots, N[\![t'_n]\!]\}$。 変換後の項の深さは
$N[\![t_0]\!] + max\{\{N[\![t'_1]\!] + max\{N[\![t_1]\!], \ldots, N[\![t_n]\!]\}\},$
$\ldots, \{N[\![t'_n]\!] + max\{N[\![t_1]\!], \ldots, N[\![t_n]\!]\}\}$。よって、変換前後で項の深さは増加しない。

＜分岐項が let 式 $t = \mathbf{let} v = t'_0\ \mathbf{in}\ t'_1$ の場合＞ (11), (13) の規則によって変換される。変換前の項の深さは
$max\{N[\![t'_0]\!], N[\![t'_1]\!]\} + max\{N[\![t_1]\!], \ldots, N[\![t_n]\!]\}$。変換後の項の深さは
$max\{N[\![[\![t'_0]\!]]\!], \{N[\![t'_1]\!] + max\{N[\![t_1]\!], \ldots, N[\![t_n]\!]\}\}$。よって、変換前後で項の深さは増加しない。

＜場合 5（let 式）:$t = \mathbf{let}\ v = t_0\ \mathbf{in}\ t_1$＞ let 式は変換規則 (10), (12) によって変換される。変換前の項の深さは $max\{N[\![t_0]\!], N[\![t_1]\!]\}$。変換後の項の深さは
$max\{N[\![[\![t_0]\!]]\!], N[\![[\![t_1]\!]]\!]\}$。よって、変換前後で項の深さは増加しない。

（証明終り）
以上のことから次のことがいえる。

定理 4.3.1 (先行評価向け切り払い手続きの停止性) 先行評価向け切り払い手続きは停止する。

(証明) 補題によって変換規則を適用する過程で、初期の深さを越える関数の展開の規則を適用することがなく、他の規則を適用する時は、項の深さが深くなるこ とがないことが分かった。プログラムにおいて、関数の定義は有限である。そして、有限の中で関数の組合せは有限である。関数記号の数が増加しないのでいつかは今までに出現したものとパターンマッチするはずである。その時は、新関数を導入して変換は終了する。よって、変換は停止する。
(証明終り)

4.3.3 変換の正当性

ここでは、先行評価向け印付き切り払いにおいて、変換の正当性が成り立つことを示す。

定理 4.3.2 (先行評価向け印付き切り払い手続きの正当性) 先行評価向け印付き切り払い手続きにおいて、変換の正当性は保証される。

(証明) 各変換規則ごとに、$t \trianglerighteq [\![t]\!]$ が成り立つことを示す。その前に、次のことを前提として証明を進める。
関数を切り払った後は、3.1 節で述べたように、変換前の関数の変数の集合と等しい引数を持ち、すべての変数が絶対評価位置にあるような関数を定義することによって変換は終了する。つまり、変換規則を初めて適用する時に存在する変数は、変換の過程において常に **let** 式で先に評価されているものと考えることができる。さらに、変換途中で新たに現れる変数も、**let** 式で先に評価されている、と考えることができる。そのように考えると、変数に停止しない項が入った時は、変換前の項も変換後の項も共に停止しないようになっている。したがって、変数に停止する項が入った時のみについて正当性が成り立つことを示せばいい。さらに、変数に停止する項が入った場合、それらを評価するステップ数は変換前も変換後も同じである。よって、変数には値（value）が入るものとして証明を行なう。 以下の証明では、s_iを基底項、d_iを値とする。そして、項 t に対する基底代入σ_iに対し、$t_i\sigma_i = s_i$の時、$[\![t_i]\!]\sigma_i = s_i'$と表す。

＜変換規則 (1) の場合＞ 何も変換をしないので改善関係は成り立つ。

＜変換規則 (2) の場合＞ 変数を含まない時は $T[c] = c$。何も変換をしないので改善関係が成り立つ。変数を含む時は、部分項に関して改善関係が成り立つと仮定する。つまり、$\forall i \in 1, \ldots, k$に対し、$t_i \trianglerighteq T[t_i]$ と仮定する。

$c(t_1, \ldots, t_k)$ は $c([\![t_1]\!], \ldots, [\![t_k]\!])$ に変換される。$c(s_1, \ldots, s_k) \xrightarrow{n} c(d_1, \ldots, d_k)$ の時、仮定より $c(s'_1, \ldots, s'_k) \overset{\leq n}{\Rightarrow} c(d_1, \ldots, d_k)$ となる。構成子式の部分項 s_i が停止しない場合は仮定より、s'_iが停止しない。よって、変換後の項も停止しない。以上より、$c(t_1, \ldots, t_k) \trianglerighteq c(T[t_1], \ldots, T[t_k])$ が成り立つ。

＜変換規則 (3) の場合＞ 次の 2 つの場合が考えられる。

＜場合 3a：変数が複数回出現するもののみである＞ この場合、関数はひとまず変数に置き換えられて変換が終了する。その変数の部分には変換前の関数が入っている。よって、変換の正当性が成り立つ。

＜場合 3b：関数を展開する＞ $f(t_1, \ldots, t_k)$ は、$t[t_1/x_1, \ldots, t_k/x_k]$ に変換される。t_i が変数の場合、値が入ることになっている。引数のところに、関数が入っている場合、$\oplus$に印付けられた関数なので全関数で、さらに線形である。よって、複数回現れるところは値が入っているので計算が複製されない。つまり、展開する関数の引数のところは、すべて有限ステップで停止する。

$f(s_1, \ldots, s_k) \xrightarrow{n} f(d_1, \ldots, d_k)$

$\rightarrow t[d_1/x_1, \ldots, d_k/x_k]$ とする。

変換後の関数においてすべての変数 $(x_1, \ldots, x_k)$ が絶対評価位置にあるとする。$t[d_1/x_1, \ldots, d_k/x_k] \xrightarrow{m} d$ の場合、変換前の項は $n + m + 1$ ステップで正規形に到達する。変換後は

$t[s_1/x_1, \ldots, s_k/x_k] \xrightarrow{n+m} d$ つまり、変換前より 1 ステップ少なくなっている。

$t[d_1/x_1, \ldots, d_k/x_k]$ の評価が停止しない場合、$t[s_1/x_1, \ldots, s_k/x_k]$ の評価も停止しない。

以上より、展開の規則を使って変換した場合、改善関係が成り立ち、かつ停止する場合は、ステップ数に関して、1 ステップ効率が良くなる。

＜変換規則 (4) の場合＞ 部分項に関して改善関係が成り立つと仮定する。つまり、$\forall i \in 1, \ldots, k$に対し、$t'_i \trianglerighteq T[t'_i]$ と仮定する。

case v **of** $p'_1 : t'_1 | \ldots | p'_n : t'_n$は、**case** v **of** $p'_1 : [\![t'_1]\!] | \ldots | p'_n : [\![t'_n]\!]$ に変換される。v には

値が入ることになっているのでここでは$p'_i(d_1,\ldots,d_k)$とおく。変換前の項は1ステップで$t'_i[d_1/x_1,\ldots,d_k/x_k]$になる。変換後の項は1ステップで$T[t'_i][d_1/x_1,\ldots,d_k/x_k]$になる。仮定より、$t'_i[d_1/x_1,\ldots,d_k/x_k] \trianglerighteq T[t'_i][d_1/x_1,\ldots,d_k/x_k]$となるので、変換前後で改善関係が成り立つ。

＜変換規則 (5) の場合＞ $\mathbf{case}\ c(t_1,\ldots,t_k)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n$ は、
$t'_i[t_1/v_1,\ldots,t_k/v_k]\ where\ p'_i\ =\ c(v_1,\ldots,v_k)$ に変換される。
ここでは、変換規則 (5) を適用する場合について変換前後で改善関係が成り立つことを示す。ここでは、パターンの中の変数を一つにして証明する。

＜条件 a の場合＞ パターンの中の変数が絶対評価位置にあり、変数の出現が線形である。この場合、計算の複製も、停止しない項の評価されない場所への移動も起こらない。今、$c(s_1) \xrightarrow{n} c(d_1)$とする。
変換前の項は
$\mathbf{case}\ c(s_1)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n \xrightarrow{n} \mathbf{case}\ c(d_1)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n \rightarrow t'_i[d_1/v_1]$。
今、$t'_i[d_1/v_1] \xrightarrow{m} t$とする。
変換後の項において、s_1は絶対評価位置にあるのでいつかは評価される。よって、$t'_i[s_1/v_1] \overset{n+m}{\rightarrow} t$
$t'_i[d_1/v_1]$の評価が停止しない場合は、$t'_i[s_1/v_1]$の評価も停止しない。
$c(s_1)$の評価が停止しない場合は変換前は明らかに停止しない。変換後は$c(s_1)$が絶対評価位置にあるので評価は停止しない。
よって、改善関係は成り立つ。

＜条件 b の場合＞ パターンの中の変数が絶対評価位置にあるが、変数の出現が線形でない。この場合、停止しない項の評価されない場所への移動は起こらない。計算が複製されることによって効率が悪くなる可能性はある。しかしこの場合、複製されるものが値である。値の評価は0ステップなので複製されても評価のステップ数が増えることはない。このような場合の変換前の項は
$\mathbf{case}\ c(d_1)\ \mathbf{of}\ p'_1:t'_1|\ldots|p'_n:t'_n \rightarrow t'_i[d_1/v_1]$
$t'_i[d_1/v_1]$は変換後の項の形である。よって、改善関係は成り立つ。

＜条件 c の場合＞ パターンの中の変数が絶対評価位置にある場合は、条件 b の場合と同様に示せる。パターンの中の変数が絶対評価位置になくて、変数の出現が線形でない。この場合、停止しない項の評価されない場所への移動、計算が複製などによって改善関係が成り立たなくなる可能性はあるのだが、複製されるものが値の場合、値の評価は 0 ステップなので複製されても評価のステップ数が増えることはない。さらに、値の評価が停止しないことはない。このような場合の変換前の項は

$\mathbf{case}\ c(d_1)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n \quad \rightarrow t'_i[d_1/v_1]$

$t'_i[d_1/v_1]$ は変換後の項の形である。よって、改善関係は成り立つ。

＜条件 d,e の場合＞ パターンに出現する変数が絶対評価位置になくて、変数は線形である。この場合、停止することが保証されているもののみ規則を適用する。$\ominus$に印付けられた関数を含まないので、項の評価は停止する。$c(s_1) \stackrel{n}{\rightarrow} c(d_1)$ とする。変換前の項は、

$\mathbf{case}\ c(s_1)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n \stackrel{n}{\rightarrow} \mathbf{case}\ c(d_1)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n \rightarrow t'_i[d_1/v_1]$。

d_iが評価されない場合で、$t'_i[d_1/v_1]$ の評価が停止する場合、変換後の項の評価も停止して、$n+1$ ステップ少なくて済む。

d_iが評価される位置にある場合で、$t'_i[d_1/v_1]$ の評価が停止する場合、変換後の方が 1 ステップ少なくて済む。d_iに関係なく停止しない場合は変換前後共に評価は停止しない。よって、変換前後の改善関係は成り立つ。

＜変換規則 (6) の場合＞ 変換規則 6 で変換した場合、変換前も変換後も **case** 式の分岐変数のところ以外は変わらない。さらに、分岐変数のところを最初に評価する。関数の展開に関しては変換規則 3 のところで変換の正当性は示されてあるので、変換規則 6 による変換も 1 ステップステップ数が少なくなる。そして、改善関係が成り立つ。

＜変換規則 (7) の場合＞ $\mathbf{case}\ (\mathbf{case}\ t_0\ \mathbf{of}\ p_1 : t_1 | \ldots | p_m : t_m)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n$ は、$\mathbf{case}\ t_0\ \mathbf{of}\ p_1 : (\mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n) \mid p_m : (\mathbf{case}\ t_m\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)$ に変換される。変数の重なりを防ぐために、**case** 式のパターンに出現する変数はすべて異なるものとして、さらに、パターンに出現する変数は自由変数とは異なることにする。$s_0 \stackrel{n}{\rightarrow} p_i(d_1, \ldots, d_k)$ とする。変換前の式について

$\mathbf{case}\ (\mathbf{case}\ s_0\ \mathbf{of}\ p_1 : t_1 | \ldots | p_m : t_m)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n$

$\overset{n}{\to}\ \mathbf{case}\ (\mathbf{case}\ p_i(d_1, \ldots, d_k)\ \mathbf{of}\ p_1 : t_1 | \ldots | p_m : t_m)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n$

$\to\ \mathbf{case}\ t_i[d_1/x_1, \ldots, d_k/x_k]\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n$

変換後の式について

$\mathbf{case}\ s_0\ \mathbf{of}$

$p_1\ :\ (\mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)$

$p_m\ :\ (\mathbf{case}\ t_m\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)$

$\overset{n}{\to}\ \mathbf{case}\ p_i(d_1, \ldots, d_k)\ \mathbf{of}$

$p_1\ :\ (\mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)$

$p_m\ :\ (\mathbf{case}\ t_m\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)$

$\to\ (\mathbf{case}\ t_i\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)[d_1/x_1, \ldots, d_k/x_k]$

今、$x_1, \ldots, x_k$はt_iにしか出現しないので

$=\ \mathbf{case}\ t_i[d_1/x_1, \ldots, d_k/x_k]\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n$

s_0の評価が停止しない場合は共に停止しない。以上より、変換の正当性が成り立つ。

＜変換規則 (8) の場合＞ 部分項に関して改善関係が成り立つと仮定する。つまり、$\forall i \in 1, \ldots, k$に対し、$t_i \trianglerighteq T[t_i]$ と仮定する。

$f(t_1, \ldots, t_k)$ は $f([\![t_1]\!], \ldots, [\![t_k]\!])$ に変換される。$f(s_1, \ldots, s_k) \overset{n}{\to} f(d_1, \ldots, d_k)$ の時、仮定より

$f(s'_1, \ldots, s'_k) \overset{\leq n}{\Rightarrow} f(d_1, \ldots, d_k)$ となる。また、部分項 s_i が停止しない場合、仮定より、s'_iも停止しない。よって、共に停止しない。

以上より、$f(t_1, \ldots, t_k) \trianglerighteq f(T[\![t_1]\!], \ldots, T[\![t_k]\!])$ が成り立つ。

＜変換規則 (9) の場合＞ 部分項に関して改善関係が成り立つと仮定する。つまり、$\forall i \in 1, \ldots, k$に対し、$t_i \trianglerighteq T[t_i]$ と仮定し、さらに、$\forall i \in 1, \ldots, n$ に対し、$t'_i \trianglerighteq T[t'_i]$ と仮定する。

$\mathbf{case}\ f(t_1^{\ominus}, \ldots, t_k^{\ominus})^{\ominus}\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n$ は変換規則によって

$\mathbf{case}\ f([\![t_1]\!], \ldots, [\![t_k]\!])\ \mathbf{of}\ p'_1 : [\![t'_1]\!] | \ldots | p'_n : [\![t'_n]\!]$ と変換される。

case 式の分岐変数の位置にある評価に関しては、仮定により変換前後で改善関係が成り立つ。つまり、$f(s_1, \ldots, s_k) \overset{n}{\to} f(d_1, \ldots, d_k)$ の時、

仮定より $f(s'_1, \ldots, s'_k) \overset{\leq n}{\Rightarrow} f(d_1, \ldots, d_k)$ となる。ここからの $f(d_1, \ldots, d_k)$ の評価は同じである。

$f(d_1, \ldots, d_k)$ の評価が停止しない時は、変換前後共に停止しない。

今、$f(d_1, \ldots, d_k) \xrightarrow{m} p'_i(d'_1, \ldots, d'_l)$ とする。

この時、変換前の項は $\mathbf{case}\ p'_i(d'_1, \ldots, d'_l)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n\ \rightarrow t'_i[d'_1/x_1, \ldots, d'_l/x_k]$ で、変換後の項は、

$\mathbf{case}\ p'_i(d'_1, \ldots, d'_l)\ \mathbf{of}\ p'_1 : [\![t'_1]\!] | \ldots | p'_n : [\![t'_n]\!]\ \rightarrow [\![t'_i]\!][d'_1/x_1, \ldots, d'_l/x_k]$ となる。仮定より、$t'_i[d'_1/x_1, \ldots, d'_l/x_k]\ \trianglerighteq\ [\![t'_i]\!][d'_1/x_1, \ldots, d'_l/x_k]$ が成り立つ。

よって変換規則 (9) を適用する前後で改善関係が成り立つ。

＜変換規則 (10) の場合＞ 部分項 t_0, t_1に関して変換の改善関係が成り立つと仮定する。

つまり、$t_0\ \trianglerighteq [\![t_0]\!]$, $t_1\ \trianglerighteq [\![t_1]\!]$ とする。$s_0 \xrightarrow{n} d_0$とする。この時、変換前の項は

$\mathbf{let}^{\ominus}\ v^{\ominus}\ =\ s_0\ \mathbf{in}\ t_1\ \xrightarrow{n}\ \mathbf{let}^{\ominus}\ v^{\ominus}\ =\ d_0\ \mathbf{in}\ t_1\ \rightarrow\ t_1[d_0/v]$

変換後の項は仮定より

$\mathbf{let}^{\ominus}\ v^{\ominus}\ =\ s'^{\ominus}_0\ \mathbf{in}\ [\![t_1]\!]\ \overset{\leq n}{\Rightarrow}\ \mathbf{let}^{\ominus}\ v^{\ominus}\ =\ d_0\ \mathbf{in}\ [\![t_1]\!]\ \rightarrow\ [\![t_1]\!][d_0/v]$。

仮定より $t_1[d_0/v]\ \trianglerighteq\ [\![t_1]\!][d_0/v]$。

s_0の評価が停止しない場合は、仮定より $[\![s_0]\!]$ の評価も停止しない。よって、変換前後で共に停止しない。

以上より、変換 (10) を適用する前後で改善関係が成り立つ。

＜変換規則 (11) の場合＞ 変数の重なりはないものとする。部分項 t_0に関して変換の改善関係が成り立つと仮定する。つまり、$t_0 \trianglerighteq [\![t_0]\!]$ とする。今、$s_0 \xrightarrow{n} d_0$とする。変換前の項は

$\mathbf{case}\ (\mathbf{let}^{\ominus}\ v^{\ominus}\ =\ s_0\ \mathbf{in}\ t_1)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n\ \xrightarrow{n}\ \mathbf{case}\ (\mathbf{let}^{\ominus}\ v^{\ominus}\ =\ d_0\ \mathbf{in}\ t_1)\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n\ \rightarrow\ \mathbf{case}\ t_1[d_0/v]\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n$

変換後の項は、仮定より $\mathbf{let}^{\ominus}\ v^{\ominus}\ =\ s'_0\ \mathbf{in}\ \mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n \overset{\leq n}{\Rightarrow}$ $\mathbf{let}^{\ominus}\ v^{\ominus}\ =\ d_0\ \mathbf{in}\ \mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n\ \rightarrow (\mathbf{case}\ t_1\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)[d_0/v]$。

いま、変数 v は分岐変数のところにしかないので

$=\ \mathbf{case}\ t_1[t'_0/v]\ \mathbf{of}\ p'_1 : t'_1 | \ldots | p'_n : t'_n)$。

s_0の評価が停止しない時は仮定より s'_0の評価も停止しない、よって変換前後の項は共に停止しない。

以上より、変換前後で改善関係が成り立つ。

＜変換規則 (12) の場合＞ 変換規則 (10) の場合と同様にして示せる。

＜変換規則 (13) の場合＞ 変換規則 (11) の場合と同様にして示せる。

（証明終り）

4.4 Let 式の取り扱い

let 式は、変換終了後に let 式を含まない形に戻す。本節では、その戻し方について説明する。

4.4.1 マイナスに印付けられた項について

let 式で分割した項を一つにまとめる時、[16] では、in の右側の項において変数の出現が複数の場合のみ、新関数を導入していた。しかし、3 章で述べたように case 式において分岐以降に対象となる変数が現れる場合、新関数を導入しなければいけない。ただし、let 式で取り出した項において関数式 (6) が含まれている場合に限る。さらに、(6) の場合でも、原始関数は全関数なので対象から外す。

例 4.4.1 次のような場合は、新関数を導入する。

$$\begin{aligned}&\textbf{let } w \ = \ f(2) \textbf{ in case } x \textbf{ of}\\ &\qquad\qquad\qquad\qquad true \ : \ w\\ &\qquad\qquad\qquad\qquad false \ : \ 3\end{aligned}$$

新関数は $g(x, f(2))$

$$\begin{aligned}&where g(x, w) \ = \ case \ x \textbf{ of}\\ &\qquad\qquad\qquad\qquad true \ : \ w\\ &\qquad\qquad\qquad\qquad false \ : \ 3\end{aligned}$$

4.4.2 プラスに印付けられた項について

let 式で取り出した項が$\oplus$の場合、in の左側の項は最終的に変換を始める時に存在したすべての引数を含む関数にする。そして、let 式を元に戻す。元に戻す方法は$\ominus$の場合と同じである。

4.5 新関数の導入について

この節では、複数回現れる変数を含む関数の変換時の、新関数の導入の方法について説明する。新関数を定義する時に、共通する計算を関数の引数としてとることによって、複数の計算が一階ですむようになる。例を用いて、関数定義の方法を説明する。

例 4.5.1 (再帰関数を導入する必要がない場合) 次の関数定義のもとでの切り払いを考える。

$f(x) \ = \ cons \ flip(x) \ cons \ flip(x) \ cons \ flip(x) \ Nil$

$flip(xt) \ = \ \mathbf{case} \ xt \ \mathbf{of}$

$\qquad\qquad\qquad Leaf \ z \ : \ Leaf \ z$

$\qquad\qquad\qquad Branch(yt, zt) \ : \ Branch(\ flip(zt), flip(yt))$

今、$f(flip(xt))$ という関数の切り払いを考える。

$T[f(\ flip(xt))]$

$= \ T[\mathbf{let} \ w = filp(x) \ \mathbf{in} \ f(w)]$

$= \mathbf{let} \ w = T[flip(x)] \ \mathbf{in} \ T[f(w)]$

3.4 節で説明したように

$T[flip(xt)] = h(xt)$

where $h(x) \ = \ \mathbf{case} \ x \ \mathbf{of}$

$\qquad\qquad\qquad\qquad Leaf \ z \ : \ Leaf \ z$

$\qquad\qquad\qquad\qquad branch(yt, zt) \ : \ Branch(h(zt), h(yt))$

$T[f(w)]$

$= \ T[cons \ flip(w) \ cons \ flip(w) \ cons \ flip(w) \ Nil]$

$= \ cons \ T[flip(w)] \ T[cons \ flip(w) \ cons \ flip(w) \ Nil]$

ここで $T[flip(w)]$ を評価するのだが、w は 2 度以上出現するものなので展開しない。

$T[flip(w)] = w_1$として変換を続ける。

$= \ cons \ w_1 \ cons \ T[flip(w)] \ T[cons \ flip(w) \ Nil]$

ここで T[flip(w)] を評価するのだが、w は 2 度以上出現するものなので展開しない。

$T[flip(w)] = w_2$として変換を続ける。

$= \ cons \ w_1 \ cons \ w_2 \ cons \ T[flip(w)] \ T[Nil]$

ここで $T[flip(w)]$ を評価するのだが、w は 2 度以上出現するものなので展開しない。
$T[flip(w)] = w_3$として変換を続ける。
$= \ cons\ w_1\ cons\ w_2\ cons\ w_3\ Nil$
ここで、$T[f(w)]$ の変換が終了した。全体を見ると次のようになっている。
$\mathbf{let}\ w = h'(x)\ \mathbf{in}\ cons\ w_1\ cons\ w_2\ cons\ w_3\ Nil$
変換が終了したので今度は引数としてどれだけとれるのかを考える。
今、w の環境は次のような状態になっている。
$[(w_1, flip(w)), (w_2, flip(w)), (w_3, flip(w))]$
このリストを見ると、共通するのは $flip(w)$ である。
つまり、$flip(w)$ を引数にとることができる。
引数として flip(w) つまり flip(h'(x)) をとって残りの部分を関数の右辺の部分に返す。
新しい変数を w とすると $w_1 \to w, w_2 \to w, w_3 \to w$ として変換を終了する。変換後の関数は $T[f(flip(xt))] = h(flip(h'(x)))$
where $h(w) \ = \ cons\ w\ cons\ w\ cons\ w\ Nil$

例 4.5.2 (再帰関数を導入する必要がある場合) 次の関数定義のもとでの切り払いを考える。

$$
\begin{aligned}
f &: (int\ list\ *\ int\ list) \to int\ list \\
f(x,y) &= g(x,y,y) \\
g &: ((int\ list)\ list\ *\ int\ list\ *\ int\ list) \to int\ list \\
g(x,y,w) &= \mathbf{case}\ xs\ \mathbf{of} \\
&\quad Nil\ : Cons(h(y), (Cons(h(y), Cons(h(z), (Cons(z, Nil))))))) \\
&\quad Cons(z, zs)\ : Cons(z, g(zs, y, w)) \\
h &: int\ list \to int\ list \\
h(xs) &= \mathbf{case}\ xs\ \mathbf{of} \\
&\quad Nil\ : Nil \\
&\quad Cons(z, zs)\ : Cons(z, h(zs))
\end{aligned}
$$

今、$f(x, h(y))$ という関数の切り払いを行なう。まず、関数 f を展開するのだが、展開すると計算が複製される部分 $(h(y))$ を別の変数に置き換えて展開する。つまり、$T[f(x,w)]$ を実行する。変換規則を適用できなくなった最終状態は、次のようになる。ここで、関数 g を変換した新関数を g' とする。

$T[f(x, h(y))]$

$T[\mathbf{let}\ w\ =\ h(y)\ \mathbf{in}\ f(x, w)]$

$= \mathbf{let}\ w\ =\ T[h(y)]\ \mathbf{in}\ T[f(x, w)]$

$T[h(y)]$

$=\ T[\mathbf{case}\ y\ \mathbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z, zs)\ :\ Cons(z, h(zs))]$

$=\ \mathbf{case}\ y\ \mathbf{of}$

$\quad Nil\ :\ T[Nil]$

$\quad Cons(z, zs)\ :\ T[Cons(z, h(zs))]$

$=\ \mathbf{case}\ y\ \mathbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z, zs)\ :\ Cons(T[z], T[h(zs)])$

$=\ \mathbf{case}\ y\ \mathbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z, zs)\ :\ Cons(z, T[h(zs)])$

ここで、再帰関数を導入して変換を終了する。

$T[h(y)]\ =\ h'(y)$

where $h'(y)\ =\ \mathbf{case}\ y\ \mathbf{of}$

$\quad Nil\ :\ Nil$

$\quad Cons(z, zs)\ :\ Cons(z, h'(zs))$

$T[f(x, w)]$

$= T[g(x, w, w)]$

$= T[\mathbf{case}\ \ x\ \mathbf{of}$

$\quad nil\ :\ Cons(h(w), (Cons(h(w), Cons(h(w), (Cons(w, Nil))))))$

$\quad cons(z, zs)\ :\ Cons(z, g(zs, w, w))]$

$= \mathbf{case}\ \ x\ \mathbf{of}$

$\quad nil\ :\ T[Cons(h(w), (Cons(h(w), Cons(h(w), (Cons(w, Nil))))))]$

$\quad cons(z, zs)\ :\ T[Cons(z, g(zs, w, w))]$

$=$ **case** x **of**

$nil\ :\ Cons(T[h(w)], T[(Cons(h(w), Cons(h(w), (Cons(w, Nil)))))])$

$cons(z, zs)\ :\ Cons(T[z], T[g(zs, w, w))]$

$=$ **case** x **of**

$nil\ :\ Cons(w_1, (Cons(T[h(w)], T[Cons(h(w), (Cons(w, Nil)))])))$

$cons(z, zs)\ :\ Cons(z, T[g(zs, w, w))])$

$=$ **case** x **of**

$nil\ :\ Cons(w_1, (Cons(w_2, Cons(T[h(w)], T[(Cons(w, Nil))]))))$

$cons(z, zs)\ :\ Cons(z, T[g(zs, w, w))])$

$=$ **case** x **of**

$nil\ :\ Cons(w_1, (Cons(w_2, Cons(w_3, (Cons(T[w], T[Nil]))))))$

$cons(z, zs)\ :\ Cons(z, T[g(zs, w, w))])$

$=$ **case** x **of**

$nil\ :\ Cons(w_1, (Cons(w_2, Cons(w_3, (Cons(w_4, Nil)))))))$

$cons(z, zs)\ :\ Cons(z, T[g(zs, w, w))])$

ここで、再帰関数を導入して変換を終了する。現在の w の環境は次のとおりである。

$[(w_1, h(w)), (w_2, h(w)), (w_3, h(w)), (w_4, w)]$

$g(x, y, v)$ の切り払いを行なった時の変換規則適用後は次の形をしている。

case x **of**

$nil\ :\ Cons(y_1, (Cons(y_2, Cons(v_1, (Cons(v_2, Nil))))))$

$cons(x, xs)\ :\ Cons(z, g'(zs, y, v))$

この時の y,v の環境は

$[(y_1, h(y)), (y_2, h(y))]$

$[(v_1, h(v)), (v_2, v),]$

上の case 式と下の case 式はマッチングするので新関数に畳み込むことができる。ここで、w の環境は

y の位置にある $[(w_1, h(w)), (w_2, h(w))]$

z の位置にある $[(w_3, h(w)), (w_4, w)]$

に分割することができる。共通部分を取り出すと y の位置にある w—h(w)

z の位置にある w—w
となり、残りの部分をもとの位置の戻して新関数を定義すると、

$g'(x, w_7, w_8)$
where $g'(x, y, v) \ = \ \mathbf{case}\ x\ \mathbf{of}$
$\qquad\qquad\qquad\qquad Nil \ : \ Cons(y, (Cons(y, Cons(h(v), (Cons(v, Nil))))))$
$\qquad\qquad\qquad\qquad cons(x, xs) \ : \ Cons(z, g'(zs, y, v))$

新しい w の環境は $[(w_7, h(w)), (w_8, w)]$ となる。ここで、全体を見ると次のようになっている。$\mathbf{let}\ w \ = \ h'(y)\ \mathbf{in}\ g'(x, w_7, w_8)$
ここで、**let** 式を戻すのだが、**in** の右側には w の出現が 2 個あるので新関数を導入する。w の環境を見ると、共通するのは w だけなので w を関数の引数としてとる。

$T[f(x, h(y))] \ = \ f'(x, h'(y))$
where $f'(x, w) \ = \ g'(x, h(w), w)$
where $g(x, y, v) \ = \ \mathbf{case}\ x\ \mathbf{of}$
$\qquad\qquad\qquad\qquad Nil \ : \ Cons(y, (Cons(y, Cons(h(v), (Cons(v, Nil))))))$
$\qquad\qquad\qquad\qquad cons(x, xs) \ : \ Cons(z, g'(zs, y, v))$

4.6 すべての変換が終了した後で

すべての変換が終了した後は、定義 2.3.1 で定義した項が組み合わさった形をしている。変換の正当性において、変数は **let** 式を用いて先に評価することを仮定して行なっている。変換終了時に、変数が絶対評価位置にない場合は、新関数を導入する。

例 4.6.1 次の関数定義のもとでの切り払いを考える。

$f(x) \ = \ cons\ x\ cons\ x\ Nil$
$g(x, y, z) \ = \ \mathbf{case}\ x\ \mathbf{of}$
$\qquad\qquad\qquad\qquad true \ : \ cons\ y\ Nil$
$\qquad\qquad\qquad\qquad false \ : \ cons\ z\ Nil$

$f(g(x, y, z))$ という関数の切り払いを考える。

$[\![f(g(x,y,z))]\!]$

$= [\![\mathbf{let} w \ = \ g(x,y,z) \ \mathbf{in} \ f(w)]\!]$

$= \mathbf{let} w \ = \ [\![g(x,y,z)]\!] \ \mathbf{in} \ [\![f(w)]\!]$

$= \mathbf{let} w \ = \ [\![\mathbf{case} x \ \mathbf{of}$

$true \ : \ cons \ y \ Nil$

$false \ : \ cons \ z \ Nil]\!]$

$\mathbf{in} \ [\![cons \ w \ cons \ w \ Nil]\!]$

$= \mathbf{let} w \ = \ \mathbf{case} x \ \mathbf{of}$

$true \ : \ [\![cons \ y \ Nil]\!]$

$false \ : \ [\![cons \ z \ Nil]\!]$

$\mathbf{in} \ cons \ [\![w]\!] \ [\![cons \ w \ Nil]\!]$

$= \mathbf{let} w \ = \ \mathbf{case} x \ \mathbf{of}$

$true \ : \ cons \ [\![y]\!] \ [\![Nil]\!]$

$false \ : \ cons \ [\![z]\!] \ [\![Nil]\!]$

$\mathbf{in} \ cons \ w \ cons \ [\![w]\!] \ [\![Nil]\!]$

$= \mathbf{let} w \ = \ \mathbf{case} x \ \mathbf{of}$

$true \ : \ cons \ y \ Nil$

$false \ : \ cons \ z \ Nil$

$\mathbf{in} \ cons \ w \ cons \ w \ Nil$

let 式の戻し方は前節で説明した通りである。

$h(\mathbf{case} x \ \mathbf{of}$

$true \ : \ cons \ y \ Nil$

$false \ : \ cons \ z \ Nil)$

where $h(w) \ = \ cons \ w \ cons \ w \ Nil$

この場合、y,z が絶対評価位置にない。この場合、新関数を定義する。

$h'(x,y,z) \ = h(\mathbf{case} x \ \mathbf{of}$

$true \ : \ cons \ y \ Nil$

$false \ : \ cons \ z \ Nil)$

where $h(w) \ = \ cons \ w \ cons \ w \ Nil$

第 5 章

まとめ

本研究では、先行評価言語における変換の正当性を満たす切り払いとして、先行評価向け印付き切り払いという新しい変換方法を提案した。さらに、関数定義の右辺における変数の線形の制限も外した形で切り払いが可能であることを示した。そして、先行評価の評価方式の特徴を利用して、積極的に中間データを作ることによって効率を良くすることができることも示せた。

今回は、$\oplus$に印付けられた項は全関数に制限した。展開の規則によって **case** 式の分岐以降へ部分関数が行くような変換が起こるためである。全関数の制限を外すことについては、関数を展開する時に、**case** 式の分岐以降へいく関数も、線形でない項を **let** 式を使って取り出したのと同様の操作を行なうことによって解決できるのではないかと思われる。また、変換規則 5 において、規則を適用できない場合でも **let** 式を使うことによって変換を続けることが可能である。

今後の課題としては次のことがあげられる。遅延評価言語を対象とした切り払いでは、高階関数に対応した切り払いも数多く提案されている。先行評価で高階関数にも対応した切り払いはまだ提案されていない。評価戦略の違いから高階関数の切り払いの問題点を考察するのも興味深い課題である。

謝辞

本研究及び学生生活全般に渡って多大なる御指導を頂いた外山芳人教授、鈴木太朗助手、そして本研究に関する議論の場を与えて下さった外山研究室の皆様に心から感謝します。

また、JAIST 食堂のアルバイトでは、研究から離れて楽しい時間を過ごすことができました。アルバイトの仲間達と厨房のスタッフの皆様に心から感謝します。

最後に、様々な形で学生生活を支えてくれた家族に心から感謝します。

参考文献

[1] Burstall,R.M. & Darlington,J., "A transformation system for developing recursive programs", J.ACM 24(1), pp.44-67, 1977.

[2] Chin,W.-N., "Safe fusion of functional expressions II: Further improvements", Journal of Functional Programming 4(4), pp.515-555, October 1994.

[3] A.P.Ershov, "Mixed Computation : Potential Applications and Prob lems for Study", Theoretical Computer Science 18, pp.41-67, 1982.

[4] A.B.Ferguson & P.L.Wadler, "When will deorestation stop?", In Proceedings of the Glasgow Workshop on Functional Programming, Glasgow Computer Science Department Research Report 89/R4, 1988.

[5] 川本史生, 中村充司, 小西善二郎, 湯浅能史, 二村良彦, "プログラム変換と数式処理システムの融合", 日本ソフトウェア科学会第 14 回大会論文集, pp.573-576, 1997.

[6] John C.Mitchell, "Foundations for Programming Languages", MIT Press, 1998.

[7] 村島康哲, "先行評価言語のプログラム変換の正当性", 平成 10 年度電気関係学会北陸支部連合大会講演論文集, pp.261, 1998.

[8] 尾上能之, 胡振江, 武市正人, "HYLO システムによるプログラム融合変換の実現", 日本ソフトウェア科学会第 14 回大会論文集, pp.447-480, 1997.

[9] Pettorossi,A. & Proietti,M., "Rules and Strategies for Transforming Functional and Logic Programs", ACM Compt. Surveys 28, pp.360-414, 1996.

[10] 佐賀 正芳 "関数型プログラムにおけるプログラム変換の研究", 北陸先端科学技術大学院大学修士論文, 1998.

[11] David Sands, "Proving the correctness of recursion-based automatic program transformations", Theoretical Computer Science 167, pp.193-233, 1996.

[12] David Sands, "Total correctness by local improvement in the transformation of functional programs", ACM Transactions on Programming Languages and Systems 18, pp.175-234, 1996.

[13] Seidl,H., "Integer Constraints to Stop Deforestation", Proc.ESOP '96, Lectue Notes in Computer Science 1058, pp.326-340, 1996.

[14] Sϕrensen,M.H., "A Grammar-based Data-flow Analysis to Stop Deforestation", Proc.CAAP '94, Lectue Notes in Computer Science 787, pp.335-351, 1994.

[15] V.F.Turchin, "The Concept of a Supercompiler", ACM Transactions on Programming and Systems 8, pp.292-325 1986.

[16] Wadler,P.L., "Deforestation: transforming programs to eliminate trees", Theoretical Computer Science 73, pp.231-248, 1990.